Temporal Reasoning for Events in a Knowledge-Based Network

Clay Stevens

A Dissertation submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Master of Science in Computer Science

2009

Declaration

I, the undersigned, declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Clay Stevens

Dated: September 9, 2009

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Clay Stevens

Dated: September 9, 2009

Temporal Reasoning for Events in a Knowledge-Based Network

Clay Stevens

Supervisor: Declan O'Sullivan

Assistant Supervisor: John Keeney

In this project, we propose a new approach to composite event detection for complex event processing using predicate matching over historical event data in a knowledge-based publish/subscribe system. The project is motivated as a way to bridge the gap between the functionality provided by semantically-enhanced publish/subscribe systems and the requirements of complex event processing systems, particularly in order to enable a knowledge-based network to perform useful event correlation.

Our proposed extension adds two major components to a knowledge-based system: a data store for historical event data and three new temporal operators based on J.F. Allen's interval calculus. We implement these components on a Java-based knowledge-based network, including a MySQL data store implementation and both a simple and advanced data store implementation. We test the scalability of the new operators in terms of processing time, and find that the processing time for the new operators scales at worst linearly with the number of events returned from the data store. Some of the advanced features of the advanced Oracle implementation add a significant amount of overhead while the simple Oracle implementation performs the best. We also explore the expressiveness of the new operators with two

case studies: a severe-weather reporting system and a system to dynamically change system logging levels.

Overall, we find that our extension successfully enables the knowledge-based system to detect patterns of events, enhancing its expressiveness, and that the approach we propose can be scalably implemented.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, the number of large-scale, distributed networks has increased dramatically, particularly due to the spread of the Internet and the advent of wireless sensor and RFID technologies [42, 25]. These technologies utilize *event-based* systems to interoperate, which has in turn led to an increased need for real-time, *complex event processing* systems in order to filter and analyze the event stream to provide relevant information to the users. This event processing is expensive at the application layer, both in terms of the amount of computation required and in terms of the price for existing commercial solutions (such as SMARTS InCharge [60] and HP OpenView [40]). New solutions must be devised to make complex event processing affordable and easy, while maintaining the expressiveness and scalability of other distributed, event-based systems.

During the 1990s, researchers developed several event-driven systems to address the need for interoperability across networks (e.g., Field [68], Yeast [50]), primarily based on the model followed by IP multicast and generalized into the *publish/subscribe* model. The ideas taken from these systems were further extended beyond local network boundaries into the Internet scale (e.g., OMG CORBA Event Service [61]). Later research divided the field into different areas based on the selection criteria for filtering messages, each of which tries to increase the expressiveness (the ability to more usefully refine the filters) while maintaining a high performance across

large-scale deployments.

The earliest publish/subscribe systems were generally *channel-based*, in which messages published to a specific channel were delivered to all users subscribing to that channel, or *subject-* or *topic-based*, where the subscriptions were made based on a highly structured and well-known set of subject attributes [27]. While the filtering operations of these systems were very simple and easily scalable, client applications were responsible for most of their own event filtering. Later systems (e.g., Gryphon [43], Siena [11]) used a *content-based* approach to message delivery, where subscriptions could filter using the entire, structured content of the messages themselves, or *type-based* filters (Hermes [65]) which treat messages as strongly-typed data objects. These approaches provided more expressiveness in the filter, but also increased the complexity of message routing [27].

Content-based networking has provided fertile ground for further research, but the advent of many new communication, content-delivery, and search-based applications have made it even more important to deliver more expressive subscriptions while maintaining performance and ensuring that subscribers receive exactly the publications they desire. A number of recent research projects have focused on improving the scalability of content-based publish/subscribe (CBPS) systems, particularly by improving the routing and matching algorithms used to deliver and filter messages [13, 5, 45], while others have focused on improving the flexibility and expressiveness of the subscriptions to allow subscribers more control over the messages that they receive [49].

In particular, some recent research projects have been undertaken to add semantic data filtering to CBPS middleware solutions [16, 79, 53, 82, 48, 47]. In [48, 47], this semantically-enhanced version of the CBPS infrastructure is termed *knowledge-based* networking, and allows for far more flexible subscriptions by linking the message data of the underlying content-based network with the rich, structured, semantic data available in application-specific ontologies. The additional semantic data provides

better expressiveness for the subscriptions while still maintaining the performance of the delivery of messages, as shown in [49].

The message filtering provided by a content- or knowledge-based system could be particularly useful in the area of *complex event processing* (CEP), with one caveat. CEP systems must *filter* a continuous event stream and detect *composite* events, which are composed out of simple (*atomic*) events based on the relationships between those events, either logical or temporal. The publish/subscribe systems described above generally only provide their filtering on the content of single events, although some of the systems (such as Siena [11] and Yeast [50]) provide some very limited support for temporal event patterns and others (PADRES [36]) allow for some logical compositions. None of the existing systems, however, provides an appropriate blend of expressiveness and scalability.

1.1 Project Motivation

The current project is motivated as a way to bridge the gap between the functionality provided by semantically-enhanced publish/subscribe systems and the requirements of complex event processing systems, particularly in order to enable a knowledge-based network to perform useful *event correlation*. Existing event correlation systems generally group events based on some externally-defined causal network (or sometimes a spatial or temporal network), which could be easily modeled as an ontology and loaded into a knowledge-based network for routing. By using the publish/subscribe architecture for message delivery, events could more easily be filtered and grouped into composite events, so long as the publish/subscribe system in question can recognize and filter on patterns of distinct events rather than only filtering the content of a single event.

1.2 Project Aims

We propose an extension to a knowledge-based network to allow filtering across multiple events so that the network can be used to filter and detect composite events, making the system useful for complex event processing. In particular, we intend to

- Investigate using temporal subscription predicates to detect composite events,
- Evaluate the scalability of our extended system in terms of predicate matching performance, and
- Explore the expressiveness of our system with some case studies representing potential real-world uses of such a system for complex event processing.

1.3 Project Approach

Our proposed extension introduces two major components to the knowledge-based network:

- A persistent data store component for storing events as they occur, and
- Temporal operators which act as subscription predicates comparing the content of the current publication to the contents of the historical publications in the data store.

We present a design for the data store and the temporal operators (based on the interval relations proposed by Allen [2, 3]) as well as an implementation of our design using a Siena-based knowledge-based network written in Java [56]. We then analyze the performance of our implementation on three vendor-specific data store implementations, one for MySQL and two for an Oracle 11g database, and explore the expressiveness of the new operators by applying them to a number of real-world case studies, including

- A severe-weather reporting system for the mid-western region of the U.S.A., and

- A client which can dynamically adjust system logging levels based on certain event patterns.

1.4 Project Contribution

Our project attepts to validate a new approach to composite event detection in complex event processing: that of detecting composite events through predicate-matching over historical event data. We also evaluate whether that approach can be scalably implemented and explore some real-world uses for the resulting system. Furthermore, we attempt to demonstrate how a semantically-enhanced content-based system, a *knowledge-based system*, could be used for complex event processing, a field which could benefit from the addition of semantic capabilities.

1.5 Document Structure

Chapter 2 explores the current state-of-the-art in complex event processing, including composite event detection and event correlation. Chapter 3 provides some background for the project, explaining publish/subscribe systems in more detail along with the temporal reasoning which informed the design of our operators. Chapter 4 describes the design of the temporal extension to the KBN, including both the added data store component and the temporal operators themselves. The actual implementation of that design is described in Chapter 5. Chapter 6 details our evaluation framework and the data received from the benchmark test of the system and provides a brief analysis of the system. Chapter 7 explores some potential uses of the temporal extensions through a set osf real-world case studies. Finally, Chapter 8 contains our concluding remarks.

Chapter 2

State-of-the-Art

The previous chapter introduced our project, including our motivations and our specific aims. In this chapter, we summarize the current research and state-of-the-art in the area in which our project falls. The first section, Section 2.1, details the current state of complex event processing, including both event stream processing and event correlation. The next section, Section 2.2, describes in detail the detection of composite events, which is an integral part of the project described in this document.

2.1 Complex event processing

In recent years, a great deal of interest has been turned toward the area of *complex event processing*, or CEP, for reasons as diverse as financial transactions [1], strategic business decisions [59], sensor networks [25], and RFID chips [42]. Traditionally, the analysis of these complex events has been done only retrospectively, but computing technology is rapidly making the real-time analysis of complex events not only possible, but desirable [51].

During the mid- to late-1980s, various database management systems developed the ability to trigger alerts and notifications when certain conditions were met, which became known as the *event-condition-action* (ECA) paradigm [15], which came to be embedded in many subsequent object-oriented or relational database manage-

ment systems (e.g., Snoop [57], ODE [34, 35], SAMOS [33]). This paradigm was further extended to support more complex kind of events, and has now led to the field of complex event processing. In general, CEP systems must first *filter* a continuous stream of simple (or *atomic*) events and detect *composite* events formed by patterns of these atomic events, connected using either logical or temporal combinations. The detected events are then *correlated* by the system to determine causal or spatial relationships between the events and to trigger certain responses from the client system. This section summarizes the current state of complex event processing, including event stream processing and event correlation and clustering, and the following section focuses in greater depth on composite event detection.

2.1.1 Event stream processing

The first step in complex event processing is to analyze a continuous event stream and *filter* the atomic events. The stream is filtered by detecting *event patterns*, which are normally found by executing queries on the event stream written in an *event processing language* (e.g. SASE [81], Cayuga [9]). These languages must be able to perform a number of functions, including

- Detecting the occurrence (or not) of events matching certain criteria,
- Imposing temporal constraints on allowed patterns of events (usually by utilizing a *sliding window* of events from the stream), and
- Imposing value-based constraints on individual events.

As an example, the SASE language [81] provides a query language which first filters the atomic events from the stream based on their *type*, then on an arbitrary series of predicates (similar to the `WHERE` clause in SQL), and finally using a *within* clause to specify the size of the sliding window during which the event must have occurred (or not, if negations are included in the constraints) [81]. This capability was extended by SASE+ [22], which expands the language to apply *Kleene closure patterns* to the event stream, allowing the language to define much more complex predicates by comparing events in the stream to each other and matching on patterns of more

than one event rather than matching on individual events [22]. The Cayuga Event Language [9] is very similar in form to SQL, allowing users to `SELECT` specific attributes from the events that match a specific *stream expression* and publish them to a specified output stream [9].

The event processing language is typically applied to the event stream using finite state machines, or *automata*. SASE [81] and Cayuga [9], in particular, both use variants of *non-deterministic finite automata*, or NFAs, for their processing. The general process for both divides the query into successive states which are reached as an event matches each of the criteria in sequence (as though the transitions between states represent successful predicate matches). If an automaton reaches its terminal state when given a particular event as input, then that event is deemed to have matched the filter [81, 9].

Summary

Within CEP, specialized query languages called *event processing languages* are used to filter events from the event stream. These languages are generally quite capable of quickly filtering extremely large quantities of events very quickly (over 50,000 events/sec in the case of SASE [81]) using query languages over the event stream that operate in a similar manner to SQL (by means of selecting events that match certain constraints). However, the predicates available are limited to such operations as can be easily expressed in a SQL-like query language. The composite event detection systems and algorithms described in Section 2.2 seem to provide much more flexibility when detecting event patterns.

2.1.2 Event correlation

Along with the filtering of events from the event stream, CEP systems also *correlate* relevant events to compose new output events for their client systems. These event correlation systems in part attempt to determine the *root cause* of a sequence of events in order to more effectively report the event occurring within the stream.

In particular, event correlation systems need to look at the causal relationships between events, which also require them to respect the temporal relationships between the two (since there is *probably* no backwards causation [7, 23, 26, 4]). The analysis given in [39] provides a more formal description of the causal relationships used to determine the correlation of events, as well as defining the logical process to be run over a causal structure to find the minimal set of causes for a particular set of events (using the SMARTS InCharge [60] system as an example). Causal relationships need not be the only types of relationships used by these systems, however, as described by Jiang and Cybenko in [46] with respect to intrusion detection. They describe a method of using a Bayesian inference network to determine correlations between multiple spatially-distinct observation spaces in order to correlate events not just according to a causal structure, but also according to their location. Similarly, they explain how to use a linear Kalman filter model to probabilistically identify possible attack correlation patterns [46].

Most of the event correlation systems so far developed are available as parts of commercial systems only (e.g., Hewlett-Packard's OpenView–now HP Business Technology Optimization solutions [40], SMARTS InCharge [60], Cisco's MARS [17]), but some systems such as SEC (the Simple Event Correlator [74]) are available for free. SEC reads events from a file stream and executes shell commands as its output stream, supporting the general requirement of correlating events and producing new ones for its clients. Event filtering is done by specifying event *contexts*, which represent the current knowledge the system has gained from its event stream. However, given its lightweight nature, SEC only supports a small number of rules, including matching a single event, a pair of events, or certain calendar-based operations. Even so, it has been successfully applied to such domains as network fault management and intrusion detection [74]. Another non-commercial system, GRACE [44], uses the open-source CLIPS tool [69] to create an expert system to perform its causal inference, demonstrating that non-commercial systems are feasible.

Summary

After filtering events from the event stream, CEP systems must also *correlate* events based on their causal (or spatial/temporal) relationships. The majority of these systems are available commercially at great expense, making them less useful for research purposes. Some free solutions (such as SEC [74], or other systems which utilize expert systems for inference [44]) are available, but only perform the event correlation without detecting composite events. There is a need for systems which can deliver composite events to such event correlation systems to complete the processing done by the commercial CEP solutions.

2.2 Composite events and subscriptions

The use of *composite* events and subscriptions has also spurred some research in recent years [73, 66, 52], which grew out of some of the ECA research done on composite events in the previous decade, such as with GEM [54] and READY [38]. These composite events describe patterns of occurrences among multiple events rather than simply containing the description of single (or *atomic*) events. Many implementations of composite events (such as that described by Li and Jacobsen [52]) allow clients to issue *composite subscriptions* for these events which include higher-order operations relating occurrences of atomic events. The higher-order operations included vary among composite event systems, but usually include operators for conjunctions of matched events, branching or disjunctive operators which match either of two events, and operators which match a specified temporal sequence of events, similar to Siena's patterns [11, 52]. This section describes some implementations of composite events, dividing the systems into those based on active databases [57, 35, 33] and those using publish/subscribe middlewares [38, 36], which are closer in nature to the system proposed in this project.

2.2.1 Active database systems

Early work on composite events was generally based on *active* database systems [35, 33, 57, 72]. Rather than acting as passive data repositories, these systems respond to particular kinds of events with specified actions. Commonly, the types of events, the conditions for activation, and the action to be executed are summarized into *event-condition-action* rules, allowing for easy response to simple events (and even some advanced processing) [64]. While there is no standardized way of representing these rules, they generally are invoked alongside modifications to the data, such as on `INSERT` or `UPDATE` SQL queries. An example of the standard ECA formulation of the rules may look something like this:

```
ON    INSERT vending_request(Money, Request)
IF    Money > 1.0
THEN  VendChocolateBar(Request)
```

These active database systems can be used to detect composite events by building up patterns of events before invoking the ECA-rules. This event composition operates in a similar manner to the event stream processing described in Sections 2.1.1, but rather than responding to a continuous stream of events, the databases only respond directly to trigger actions on the data stored within them. These simple events are generally composed into composite events using one of two approaches: either *tree-* or *graph-based* algorithms or with finite state machines (*automata*). This section describes each of these two approaches, including some specific systems which are examples of one or the other.

Tree-based systems

In tree-based composite event detection systems, composite events are defined using a tree like structure, with simple events representing the leaves of the tree and the desired composite event representing the root. The simple events are joined into composite events using various logical or temporal operators which generally fall into the following four categories:

- *Conjunctions*, where two or more events must both be detected regardless of order;
- *Disjunctions*, where any one of two or more events must be detected;
- *Sequences*, where two or more events must be detected in a specified order (sometimes with constraints on the temporal relations); or
- *Repetitions*, where a specified event happens more than once, either periodically or non-periodically.

One example of such a system is Snoop [57]. Snoop organizes events into a hierarchy including database events such as INSERTs and UPDATEs and temporal events, organized similarly to the Yeast [50] events described above with both *absolute* and *relative* events. The composite events are formed by composing simple events using disjunctions (OR), conjunctions (AND), sequences, and periodic or aperiodic repetition. The detection of composite events is done by forming a graph of activated events, which are linked together into trees in the *event forest* used by the composite event detector to find patterns of simple events which fit the definition of the composite event. Once all the simple event leaf nodes have been activated, the composite event itself is created and signaled, and the corresponding actions are performed [57].

More complicated graph-based algorithms can also be used to detect composite events from simple ones, such as the *colored Petri nets* used in the SAMOS system [33]. Petri net algorithms represent a discrete system as a directed graph, where the nodes represent *transitions* and *places* (either for input or output) within the system. *Tokens* are transported throughout the system whenever a transition is fired, which moves tokens from the input places connected to that transition into the output places of that transition [67]. In SAMOS, the simple events of the system are represented by the input places of the Petri net, the composite events represented by the output places, and the composing expressions and conditions are represented by the transitions. As the tokens travel through the output places, the corresponding composite events are signaled to the system [33].

Automaton-based systems

As an alternative to using the tree- or graph-based algorithms above, composite event detection within an active database system is sometimes done using finite state automata. One such system was proposed by Gehani, et al. [35] for the Ode database system. Ode groups events into *histories*, which are finite sets of event occurrences with unique event identifiers. *Event expressions* are represented as functions which map events from the histories in the expression domain to a new set of histories which serves as the function's range. These expressions (which provide operators for conjunction, disjunction, and temporal relationships) can be encoded into automata wherein each primitive event occurrence results in a transition from one state to another. When a composite event is triggered (by the automaton entering an accepting state), the system then builds an *occurrence tuple* out of the history to explain why the event was triggered[34]. The occurrence tuples correspond to the tree structure stored by Snoop [57] by containing not only the resulting composite events, but also each of the contributing atomic events.

2.2.2 Composite event detection in middleware

Along with the active database systems, much effort has been put into developing composite event detection middlewares (or extensions thereof) for distributed systems. Rather than relying upon ECA-rules defined within an active databse, however, these systems commonly operate as overlay networks which respond to events that are explicitly published to the system. These systems can similarly be divided into tree-based or automaton-based systems, examples of each of which are briefly described in this section. This section describes two such systems: READY [37, 38] and PADRES [36, 52].

Tree-based systems

Middleware systems which detect events using a tree-based algorithm operate in a manner very similar to the tree-based active databases discussed in the previous sub-section. In general, simple events acts as the leaf nodes of the tree and are then composed using various operators to generate tree structures with composite events as parent nodes. Subscriptions on composite events specify the tree structure which is to be realized in notifying the subscriber.

As an example, Gruber, et al. developed an event notification service called READY [37, 38] which allows for composite event detection and delivery. It also operates using something similar to the ECA paradigm described above, with event *consumers* providing *specifications* that are matched with expressions that result in specified actions. These events are matched as either *simple matching elements*, which correspond to the leaves of the tree, or *compound matching elements* which occur further up the tree as parent nodes. Any trees which match the structure of any subscriptions are disseminated through the network to the event consumers [38].

A similar algorithm is used to detect complex events in the PADRES system [36]. PADRES is of particular interest for this project because of the robust way in which composite events are used to exploit temporal relationships between events. PADRES uses *composite subscriptions* to link together the events matching the standard *atomic subscriptions* which are matched by individual events, thus allowing the middleware itself to detect the occurrence of composite events given the appropriate subscriptions. In particular, PADRES supports compositions of atomic events which represent *parallelization* or a conjunction of events, *alternation* or disjunction, *sequence*, and *repetition* [52]. Parallelization and alternation compose the events without regard to their temporal ordering, but sequence and repetition very much take it into account. These composite subscriptions are routed through the network to be as close as possible to the publishers before being decomposed (PADRES exclusively used the publish-subscribe-advertise model) [52]. This also

makes each individual broker a composite event detector, allowing the middleware itself to do complex event processing.

Automaton-based systems

As an alternative to using a basic tree structure to detect composite events within a middleware system, Pietzuch, et al., propose a generic middleware extension which can be used on top of any publish/subscribe middleware [66], based on their experience with Hermes [65]. They propose implementing a generic composite event framework to be deployed alongside an underlying publish/subscribe network (such as Hermes [65] or even Siena [11]) rather than attempting to modify the network to introduce this functionality. The composite events within this generic framework are to be modeled as patterns of the atomic events which can be detected by the underlying publish/subscribe system, which are subscribed to independently by the composite event detector itself. These composite events relate the atomic events using interval time semantics similar to Allen's [2] and are detected by finite state automata, which have two types of state: *ordinary* and *generative*. The generative states create new events out of the events matched so far or events in the future, whereas the ordinary states simply represent the state of the automaton. As new events occur, the automata change state. If a generative state is reached which has no outgoing transitions, the prescribed event is created and the automaton terminates immediately, which can result in the signaling of a composite event. The system is distributed via the decomposition and distribution of the composite event expression, and thus the decomposition of the resultant automaton, among the nodes in the network [66]. Because the system is distributed and agnostic with respect to the underlying publish/subscribe system, Pietzuch, et al., find it is an effective framework for composite event detection.

Summary

Composite events are compositions of simple or *atomic* events joined by logical or temporal operators which generally express *conjunction*, *disjunction*, *sequence*, or *repetition*. These systems can be divided between two main groups: *active databases*

(such as Snoop [57], SAMOS [33], and Ode [34]) and *middleware* systems (e.g., READY [37, 38], PADRES [36]). Those divisions can each be further sub-divided between systems which use a *tree-* or *graph-based* detection algorithm and those which use *finite-state automata*.

Active database systems generally employ *event-condition-action* rules by which they respond to data manipulation events (such as `INSERT` and `UPDATE` queries) which meet certain conditions by executing specified actions. However, this limits the active databases to responding to data manipulations rather than arbitrary events. Middleware systems get around this limitation by responding to any events that get published to them, but either sort of matching algorithm (graph- or automaton-based) requires the implementing system to continually be engaged in a form of forward matching which consumes memory by forcing the brokers to preserve the portion of each composite subscription which has been matches as state at the broker.

Rather than limit the current project to an active database system (and thus rely on the operations allowed by the ECA-rules) and sacrifice the computational resources necessary to save state for forward matching (as in existing middleware systems), we propose to detect composite events in our system using a form of *predicate matching* against *historical event records* within a *content-based, publish/subscribe* middleware. This will give our system the flexibility exhibited by systems such as PADRES [36] without requiring any forward pattern matching.

Chapter 3

Background

In the previous chapter, we described the current state-of-the-art in complex event processing, including composite event detection and event correlation. This chapter focuses on the research which led directly to the current project, particularly in terms of the knowledge-based network upon which it is based and the temporal reasoning used in our extension.

Section 3.1 provides a brief description of publish/subscribe technologies, including a discussion of content-based networking in general. Section 3.2 describes the Siena [11] project (which is extended in our implementation) in some depth, followed by Section 3.3 which discusses *knowledge-based networking*, including the extension to Siena developed by Keeney, et al.[49]. Finally, Section 3.4 summarizes the types of temporal relations which formed the basis for the operators proposed in this project.

3.1 Publish/Subscribe Systems

Publish/subscribe systems began to appear in the 1990s as a way to handle interoperability among the components of early distributed systems. The earliest of these systems (such as Field [68]) was designed following the model of IP multicast, in which networks hosts are able to dynamically join and leave *groups* identified by a single IP address within the network. Datagrams sent to this multicast address are then subsequently delivered (using a best-effort strategy) to all current members

of the multicast group [21]. Field [68] itself allowed integrated tools to send rudimentary subscriptions to a central message server, using string patterns which were matched against the messages received. Later systems refined the idea of the filters used for subscriptions and can now be divided into three main categories: *channel*- or *topic*-based, *content*-based, and *type*-based.

Channel-based systems (such as the OMG CORBA Notification Service [62] and SCRIBE [14]) allow subscribers to specify a particular *channel* or topic in which they are interested. Any message published to that channel will be delivered to any subscribers currently in the channel, making these channels very similar to basic multicast groups with the topic taking the place of the group address. This approach allows for very efficient routing (since the topics are static), but greatly reduces the expressiveness allowed for the subscriptions, since only a small set of channels will be allowed. Furthermore, many of the subscribers may need to do further refinements at the layer above the channel-based system, since the channels are very coarse-grained.

In order to allow for greater expressiveness, subsequent systems expanded the subscription method to allow for complex matching on the structured content of the messages themselves. This is referred to as *content*-based networking, which has been implemented in many systems such as Gryphon [6, 43], JEDI [18], Siena [12, 11, 13], PADRES [36, 52], and REBECA [28]. Content-based systems allow subscribers to provide *filters* on message content which then act as the address of the subscriber. As messages are published, the intervening network routers (or *brokers*) evaluate the message to see if its content matches the filters of which that broker is aware and routes the matching messages accordingly. This approach greatly enhances the expressiveness of the subscriptions, in effect allowing subscribers to define their own criteria for messages rather than relying on some externally-defined set of topics. This reduces the amount of filtering the client applications must do, but it also increases the overhead of routing the messages, since each broker must now determine how to route messages based on the subscriptions and messages it receives.

In order to combat some of the overhead required in content-based networking, a few recent systems (such as Hermes [65] and Knight [19]) have formalized the structured content of the messages into *type*-based systems. These systems utilize a robust typing system for the messages, allowing publishers to add their own event types into a type hierarchy and allowing users to subscribe to messages that inherit from specific types. This does reduce some of the overhead of the routing (since it uses standard programming concepts such as types), but the gain comes at the expense of limiting the subscriptions to the rigid type hierarchy.

The evolution of publish/subscribe systems from the initial multicast groups into the more advanced content- and type-based systems has been a very fertile research area over the last decade. This fecundity looks to continue forward from the current position, especially in refining and improving the performance of content-based systems (which allow the greatest flexibility for subscribers) and in extending the paradigm to allow for greater expressiveness. The current project is based on an extension to the content-based paradigm, *knowledge-based* networking, which will be described in further detail in Section 3.3. The following section summarizes the operation of the underlying content-based system, Siena [12].

3.2 Siena: Content-based Networking

Siena [12] is a content-based event notification service designed to provide both expressive subscriptions and to allow for Internet-scale deployments. By design, it acts as an overlay network placed on top of some underlying networking framework. The actors within the system fall into three main categories: *publishers* who introduce messages into the system, *subscribers* who consume these messages, and *brokers* responsible for routing the messages through the overlay network. Each subscriber issues a *subscription* message to a broker, which contains a *filter* which the broker applies to subsequent messages received from any of the publishers currently using the system. Messages from the publishers are called *publications*, which are routed

through the system. If a subscriber A issues a subscription S, then any publication which matches S will be delivered to A as a *notification*.

The subscription filters allowed by the original Siena brokers support four basic data types (integers, floats, character strings, and Booleans) and a small set of *operators* which define relations between values of those types, including numeric comparison operators ($<$, $\leq$, $=$, $\geq$, and $>$) and some string operators (substring, prefix, and suffix). The filters themselves consist of a set of *attribute constraints* which include the name of a publication *attribute*, an *operator*, and a *target value*. A constraint is matched by a publication, P, if the value in P of the attribute specified in the constraint satisfies the relation defined by the operator with respect to the constraint's target value. For example, a subscriber could specify a constraint of

$$\texttt{price} \quad < \quad 80$$

which would then be matched by any publication which had a `price` attribute with an integer value of less than eighty. If a publication matches every constraint in a filter, it matches that filter.

The operators used in the attribute constraints also play a large role in the routing scheme used by Siena, which makes use of *covering* relations between subscriptions to achieve a form of sub- and super-netting. For two subscriptions, S and S', S *covers* S' if and only if every publication which matches S' also matches S [12]. This information is used in the routing to group finer-grained subscriptions as subnets beneath subscriptions which cover them, decreasing the amount of routing information that must be stored by each broker. This is particularly useful if Siena is deployed on a hierarchical topology, where the covering relations can be closely modeled by the master-subserver relationship between brokers. Siena also allows for a peer-to-peer deployment, however, which also requires that publishers send *advertisements* describing the publications they will produce in order to correctly build the routing information with respect to the covering relations.

The filters described above only operate within the scope of a single publication, but Siena also includes some very basic support for capturing *patterns* of events. These patterns are composed of multiple filters which are matched in sequence, each to a corresponding event. As an example, consider a pattern composed of two filters, $F_1 \cdot F_2$. A broker evaluating a subscription on this pattern would first evaluate publications against the first filter, F_1. If a publication, P_1 matching F_1 is found by the broker, the broker will then seek to match incoming publications against the second filter, F_2. If a second publication, P_2 matches F_2 (regardless of the number of intervening publications), the broker will deliver both P_1 and P_2 to the subscriber and will resume matching publications against F_1 [12].

In practice, Siena provides an appropriate mix of expressive flexibility with the types of subscriptions allowed and good scalability across a network, mostly due to the use of the covering relations. For matching single publications, the approach is useful; however, the pattern matching capabilities are less useful. The implementation of pattern matching was incomplete, and the simplicity of the allowable patterns makes it a far less attractive solution than the more robust solutions for pattern matching described in Section 2.2. Further extensions are needed in order to more usefully support event patterns, as well as extensions which allow for more operators and collections of data. The current project proposes an extension for greater pattern support, and the following section describes a knowledge-based network built on Siena, which adds semantic operators and support for collections.

3.3 Knowledge-based Networking

The increased flexibility of the content-based paradigm can be extended even more through the use of semantic information, which has been attempted in a number of recent research projects [16, 82, 79, 48, 49, 47]. This new, semantically-enhanced paradigm is called *knowledge-based* networking [47]. These projects associate the content of the publications sent through the network with external semantic data

stored in ontologies elsewhere in the system, allowing for greater expressiveness by leveraging the semantic relationships expressed by the messages.

In particular, [49] describes an extension to Siena [12] which enhances the system in two ways: by adding the *bag* or *multiset* collection type and some associated operations (such as subbag and superbag operations), and by linking the system to a Jena ontological reasoner [41] and introducing some relevant semantic operators (such as subsumption and semantic equivalence).

The bag operators are based on the *bag* or *multiset* collection, which distinguish distinct collections based on the multiplicity but not the order of the elements, unlike standards sets which do not distinguish multiplicity. For example, the bag `{ 1, 1, 2 }` would be considered equivalent to the bag `{ 1, 2, 1 }` but not to `{ 1, 2 }`. The bag extension to Siena introduces the bag as a data type (along with the standard numeric and character string data types) and also introduces three simple bag operators which can be used in conjunction with the other operators of Siena to create composite bag operators. The three simple bag operators define bag *equality* (where each member of either bag occurs the same number of times in the other bag), the *subbag* relation (where one bag is contained within the other), and the *superbag* relation (the inverse of the subbag relation).

The more complicated *composite* bag operator (not to be confused with the composite events discussed in Section 2.2) is defined by a main operator (one of the simple bag operators) and a sub-operator, which can be any operator. The composite operator takes two bags, B_1 and B_2 as operands, and compares the two bags using the main operator with respect to the sub-operator instead of the standard equivalence relation. For example, consider the following constraint applied to a publication P, which has the bag V as the value of its `bag` attribute:

$$\texttt{bag} \quad SUBBAG_{<} \quad \{\ 1,\ 3,\ 5,\ 7,\ 9\ \}$$

When the broker applies the composite bag operator to P, it will attempt to find a subbag B of the bag specified in the constraint where each member of V is less than the corresponding member of B. The bag operators provide a large boost to the expressiveness of single subscriptions, in that certain sorts of conjunctive or disjunctive subscriptions which previously could be addressed only with multiple subscriptions can now be expressed as a relation to a bag of possible values instead of only one [49].

The extension to Siena in [48, 49] also added a set of operators used to represent semantic relations in subscriptions. These new operators express semantic equivalence, class and property sub- and supersumption, and instantiation as well as allowing the sub scriber to define an arbitrary ontological property to be used when determining whether or not an operator applies. The new semantic operators are shown in Table 3.1 along with a brief description of the relation the express.

Operator	**Description**
EQUIVAL	Semantic equivalence (`owl:equivalentClass`, etc.[76])
NOT_EQUIVAL	Semantic distinction (complement of EQUIVAL)
LESSSPEC	Subsumption (`rdfs:subClassOf`, etc.[77])
MORESPEC	Supersumption (inverse of LESSSPEC)
IS_A	Instantiation (`rdf:type` [77])
IS_NOT_A	Exclusion (complement of IS_A)
ONTPROP(P)	Arbitrary ontological relation (P)

Table 3.1: Semantic operators for a knowledge-based network [48, 49]

While the semantic and collection operators added by the knowledge-based extension do increase the flexibility of the subscriptions in Siena, they still do not provide any support for patterns of events, as these operators still only apply to one publication at a time. The current project adds this support through the use of temporal operators, the reasoning behind which is described in the following section.

3.4 Temporal Reasoning

The majority of time-based reasoning in event processing systems to date is based on the work by James F. Allen in the early 1980s [2, 3]. Allen introduces the con-

cept of representing temporal intervals, as opposed to earlier work which used only point-based representations of events [55]. Using these intervals, Allen identifies a total of thirteen unique relations which can exist between two intervals, X and Y, consisting of seven base relations and their inverses.[1] The seven base interval relations as defined by Allen are depicted in Figure 3.1.

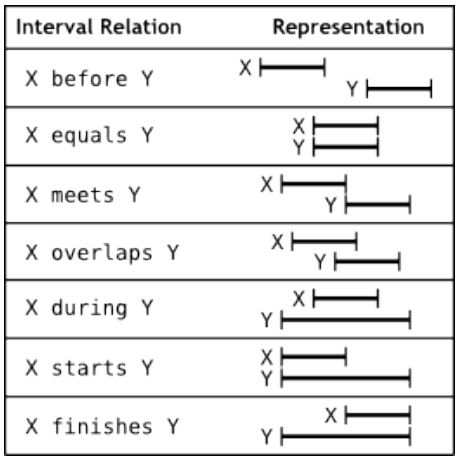

Figure 3.1: Allen's interval relations [2, 3]

These interval relations can also be described based on a comparison of their start and end points, which can be represented numerically. The interval relations as translated into numeric expressions are listed in Table 3.2 (X_S represents the beginning of interval X, and X_E represents its end).

Interval relation	**End-point relation**
X before Y	$X_E < Y_S$
X equals Y	$(X_S = Y_S) \wedge (X_E = Y_E)$
X meets Y	$X_E = Y_S$
X overlaps Y	$(X_S < Y_S) \wedge (X_E > Y_S) \wedge (X_E < Y_E)$
X during Y	$(X_S > Y_S) \wedge (X_E < Y_E)$
X starts Y	$(X_S = Y_S) \wedge (X_E < Y_E)$
X finishes Y	$(X_S > Y_S) \wedge (X_E = Y_E)$

Table 3.2: Interval relations represented as terminal-point expressions [2, 3]

Allen goes on further to describe how these relations could be used by maintaining a network of intervals based on their temporal relations (where the intervals act as nodes and the relations as the edges) which can then be maintained by applying a

[1] The "equals" relation is its own inverse.

constraint-satisfaction algorithm to compute the transitive closure of the network. Thus, when a new fact is entered into the network, the algorithm adds all the new information that can be inferred from the new fact. Uncertain relations can be represented as vectors within the network, which act as restricted disjunctions on which relation applies between two intervals, which would not be possible if the intervals were represented as zero-duration end points. However, as pointed out by Vilain, et al. [75], determining the complete transitive closure with Allen's proposed vector-based interval algebra is an NP-complete problem, which makes it intractable using modern computational techniques. Vilain, et al. suggest that restricting the algebra to those subsets which are tractable could still prove useful. Their restricted subset, which they call the *continuous endpoint algebra*, precludes all non-continuous point operations (such as non-equivalence) and any truly disjunctive relationships. Computing the transitive closure over this restricted algebra is tractable (as shown in [75]), which makes it a more attractive choice for computational use.

One of the main criticisms of such a point-based algebra is that it does not correspond to real-world events, which can sometimes have uncertain beginnings and endings rather than explicitly-declared and well-defined end points. Some approaches are to utilize fuzzy temporal relations [71, 70, 8, 24] rather than the well-defined end points of the Allen relations. Christian Freksa describes a different approach, using semi-intervals (which are the generalized "beginnings" and "endings" of intervals rather than explicit durations of events) as the basic unit of comparison, which can better handle the uncertainty about the end points [31]. However, despite the criticisms, Allen's original description of the relations is still very intuitive and serves as the ultimate basis for the temporal relations used in most event-based systems.

As an example, the Yeast (Yet another Event-Action Specification Tool) [50] system uses similar temporal relationships in order to specify the rules used to signal events based on the time of their occurrence *relative* to other events. The relative time operators they implemented (in contrast to their *absolute* operators, which use an externally defined calendar system) determine whether a certain rule should trigger

`in` a certain amount of time (and match *permanently* from then on) or `within` a specified interval (which matches only *transiently*) [50]. Similarly, Walzer, et al.[78], proposed an extension to the Rete algorithm [29] which enables it to do more advanced complex event processing by adding support for Allen's interval relations. Their extension adds new behaviors to the Rete *beta*-nodes (which are used to compose *facts* in the *working memory set* into sets which match the rules specified to the system [29]) to check the relative temporal constraints along with the standard Rete join conditions. The relations added closely match those described by Allen in [2], but include some parameterization for the DURING and EQUAL operators which allow for specifying quantitative constraints on the size of the allowable distance between end points [78].

As shown by Yeast [50] and the temporal extension to Rete [78], Allen's temporal reasoning is well-suited to performing complex event processing in an event-based system, which is precisely the approach advocated in this project. The next chapter, Chapter 4 describes the design of our proposed extension to knowledge-based networking which adds support for temporal operators inspired by Allen's interval relations as well as a data store component to maintain a store of previous publications against which the temporal operators may be applied.

Chapter 4

Design

The previous chapter summarized the background for this project, including a description of content- and knowledge-based networking and the interval relations described by Allen [2, 3]. In this chapter, we describe the design of the data store and temporal operators we propose to add as an extension to the knowledge-based networking paradigm. The chapter begins with a brief description of our approach and the motivations behind our choice, then continues with an explanation of the data store component and temporal operators before finishing with a brief summary and a discussion of the merits of the design.

4.1 Approach

In order to provide support for composite event detection to a knowledge-based system, we propose adding

- A persistent *data store* component for storing historical event data, and
- A set of *temporal operators* to be used in matching subscription predicates against the historical data stored in the data store using Allen's interval relations.

This approach stands as a contrast to the existing systems described in Chapter 2, which utilize active databases rather than distributed, publish/subscribe systems

or detect events using tree- or automaton-based forward matching. We elected to extend a knowledge-based publish/subscribe system for three main reasons:

- The active database systems reviewed are too restrictive on the types of events used in the ECA-rules, and distribution of an active database is beyond the scope of this project,
- Using a tree-algorithm or a finite state machine for forward pattern matching requires the system to save the state of each partial match for each composite subscription at each broker, which greatly increases the complexity of the matching algorithms the broker must invoke for each publication, and
- The semantic capabilities of knowledge-based systems (which are lacking in other approaches) provide a significant increase in expressiveness to the sorts of events which could be detected in complex event processing.

In summary, knowledge-based publish/subscribe systems provide us with an appropriate mix of scalability and expressiveness in a distributed system.

4.2 Data store design

In order to store historical event data for future comparison using our proposed temporal operators, our extension adds a simple data store component to each broker of the base knowledge-based system. This new component allows each broker to store, retrieve, and analyze publications as they are transferred through the framework. In particular, the data store needs to be able to perform three major functions[1]:

1. Storing (or updating) uniquely-identifiable events (as represented by publications) in some persistent data store,
2. Retrieving specific events (as publications) for inspection by the system, and

[1]The temporal ordering of the events is assumed to be represented by some attribute in the publications themselves.

3. Checking to see if the data store contains information about any events which match a particular filter given other arbitrary constraints (such as checking the value of certain attributes).

A generic interface for this component is shown in Figure 4.1.

```
<<Interface>>
  DataStore
+ getEventById(id) : Publication
+ hasMatchingEvent(Filter, ...) : boolean
+ storePublication(Publication) : boolean
```

Figure 4.1: Data store interface design

The first method, `getEventById()`, simply queries the database for the event identified by the passed identifier and returns the information stored about that event as a *publication*, which is simply a *name-type-value* triple in a knowledge-based system. This function serves to meet the second of the three requirements. The second method, `hasMatchingEvent()` compares the events in the data store to the passed filter and returns a result of `true` if any of the events match that filter, subject to the other arbitrary constraints (represented by "`...`"). This satisfies the third requirement. And finally, the last method shown in Figure 4.1, `storePublication()`, inserts the passed publication into the data store, either by creating a new event representation or by updating the event to which the publication refers.

4.3 Operator design

Along with the data store component, our extension also adds a small set of new operators to the knowledge-based system which enable it to represent and detect temporal relationships between the publication currently being examined by a broker and the historical event data already stored by that broker. These operators are inspired by the interval relations defined by Allen [2, 3], with some alterations. In particular, the processing done by our predicate-matching approach happens as soon as a publication is received, so in effect, no future information can be relied upon to

Interval Relation	Representation	Operator
X before Y	X Y	Y_S AFTER X
X equals Y	X Y	Y_S WITH X, Y_E WITH X
X meets Y	X Y	Y_S DURING X
X overlaps Y	X Y	Y_S DURING X, Y_E AFTER X
X during Y	X Y	X_S DURING Y, X_E DURING Y
X starts Y	X Y	Y_S WITH X, Y_E AFTER X
X finishes Y	X Y	X_S DURING Y, X_E WITH Y

Figure 4.2: Allen's interval relations represented as point operations

be used in determining if the operators can be applied. Furthermore, the operators only operate over single attribute values from the current publication (such as the beginning or end timestamp) compared to the entire interval (both start and end timestamps) of the previously stored publications.

Due to these additional constraints, we consider Allen's interval relations as they relate to comparing a single time stamp (either the start or end time of one interval) to another interval. Using this view, each of Allen's seven intervals can be represented using three operators representing the relationship of a point P (either a start or an end) to an interval X: $AFTER$, $WITH$, and $DURING$. Figure 4.2 shows how these operators relate to Allen's interval relations.

The three operators all operate by comparing a *reference time* X_R, either the start or end of one interval X, to a target time Y_T from the second interval Y. For the $AFTER$ operator, the reference time varies between X_S (the start of X) and X_E (the end of X), but the reference time is always the end of Y, Y_E. The $DURING$ operator actually compares X_R to *both* the start and end of Y, giving it two target

times. Finally, the $WITH$ operator compares X_R to the corresponding timestamp of Y, such that $R = T$. As shown in Figure 4.2, these three operators can represent any of Allen's seven relations.

In order to use the three operators listed above in our extension, we first need to represent them as predicates taking a *reference attribute* as the reference time and a *target filter* (which selects events from the data store) which returns events corresponding to the target interval. For $DURING$ and $AFTER$, we use a direct translation of the reference times into attributes, using the attributes X_S and X_E to represent the start and end time of a publication X, respectively, and a filter F as the target. An operator applies if there exists a stored publication, Y, such that the relation defined by the operator holds between the reference attribute and the start and end times of the event represented by Y. For performance tuning, the $AFTER$ operator accepts an additional time limit, L, which defines the size of the window between the end time of the compared event and the reference time.

The $WITH$ operator, however, can be made more general when applied to publication attributes, in that it compares the value of an arbitrary attribute of a publication X to the value of the *same* attribute in any matching stored publication, Y. By generalizing the equivalence test from $WITH$ into an arbitrary operator, we represent the $WITH$ operator described above with a general $FILTER$ operator that can compare any arbitrary attributes using an arbitrary operator. The $FILTER$ operator also accepts a configurable limit on the size of the result set (L) which is to be examined when searching for a match. The three operators added by our extension are summarized in Table 4.1 (where $F(Y)$ means that the publication Y matches the filter F, and X_R represents the reference attribute).

Operator	**Condition**
X_R **DURING** F	$\exists Y : F(Y) \wedge (Y_S \leq X_R) \wedge (Y_E \geq X_R)$
X_R **AFTER(L)** F	$\exists Y : F(Y) \wedge (Y_E < X_R) \wedge (Y_E \geq (X_R - L))$
X_R **FILTER(OP, L)** F	$\exists Y : F(Y) \wedge (X_R \, OP \, Y_R)$

Table 4.1: Operator summary

It should also be noted by any implementers of this design that any subscribers using the proposed operators would not necessarily be issuing a subscription for the target filter evaluated by the operator. If each broker operates using its own independent data store, the brokers themselves must subscribe to the inner filters in order for the broker to be able to correctly apply the new operators without missing any of the relevant publications being routed through the system. In a system such as the hierarchical model of Siena [12], each broker receiving a subscription containing one of the proposed operators must issue a subscription to its master for publications which match the inner filter alone. This only needs done for first-level inner filters, however, as the next broker up the tree should send the second-level, the next the third, and so on.

4.4 Analysis

Our aim in proposing the design for this extension was two-fold:

- To create a data store component which can be added to the brokers of a knowledge-based system which can store, retrieve, and analyze historical event data, and
- To add new operators to the knowledge-based system which can be used to relate events using Allen's interval relations.

In fulfillment of those aims, the design presented in this chapter described a data store interface capable of realizing all the data store needs for the system and three operators ($AFTER$, $DURING$, and the general $FILTER$ operator) which can represent all seven of Allen's interval relations. Furthermore, the two temporal operators very simply represent the other four (doing only two integer comparisons each) and the $FILTER$ operator, by virtue of its generality, allows for even more applications of the operator than those considered in this document.

With regards to performance, the data store must iterate over all the events which

initially match the time constraints provided by the operators, which means that the time taken to process each operator should grow as more publications are stored in the data store, up to the limits configured in the operators themselves. It should be feasible to implement this design in such a way that the processing time grows no worse than linearly with the number of events returned from the data store (such as the implementation described in Chapter 5).

As a further benefit, our design could also be employed on any content-based publish/subscribe system, not just a knowledge-based system, as long as the system can serialize a filter and pass it as the target of our operators. The actual time values used by the system do not matter (as long as they are fully-ordered and defined as publication attributes) and the data store component is simple enough to be implemented any number of ways, either with a relational database system or even using flat file storage.

Overall, we find that this design simply and completely meets all the requirements we set forth for this project. Additionally, the design is general enough to apply to any content-based publish/subscribe system using a variety of data store implementations, and the additional expressiveness provided by the $FILTER$ operator could be extended to represent many different sorts of relations between publications. The next chapter describes a proof-of-concept implementation of this design extending a Java-based knowledge-based system.

Chapter 5

Implementation

The previous chapter described the general design of our proposed extension to enable a knowledge-based publish/subscribe system to detect patterns of events based on their temporal relations. We added a data store component and three new operators: $AFTER$, $DURING$, and a generic $FILTER$ operator. In this chapter, we will describe in detail the implementation of this design that we used to evaluate the extension. This chapter will begin with a brief introduction to our implementation, including any assumptions and conventions we adopted. This will be followed by an explanation of our data store implementations and the implementations of our operators, as well as the utility classes we introduced to ease the use of our extension. Finally, we will discuss the changes that we introduced to Siena and the KBN.

5.1 Implementation description

Our implementation is based on a Java-language implementation of a knowledge-based system described in [48] which itself was implemented on top of Siena [12]. As such, our implementation utilizes the types and operators of those systems, explained in Chapter 3.

The implementation relies on a number of assumptions about the use of the system, specifically relating to the timestamps:

- Event start and end timestamps are totally-ordered and defined by some external source,
- Event timestamps are to be delivered as publication attributes,
- The end time of a particular event will always be greater than or equal to the start time of that event, and
- Any publication marked to represent the end of an event will be preceded by a publication marked to represent the start of that same event (excluding instantaneous events).

Furthermore, as a general maxim in content-based systems is to minimize the amount of meta-data attached to each publication, we have implemented our design using a number of conventions which allow it to utilize the existing Siena framework in a consistent manner. In particular, our implementation uses four specific attributes to represent the start and end times of our events, as well as the unique identifiers for the events and some additional flags to denote whether a publication represents the start or end of an event (or an instantaneous event, in which the start and end time are equal). These attributes are described in Table 5.1, and are utilized by the helper classes described in Section 5.4.

Attribute Name	**Attribute Type**	**Description**
PUB_ID	String	Event UUID
KBN_PUB_TYPE	Bag[1]	Publication type
KBN_START_TIME	Long	Event start timestamp
KBN_END_TIME	Long	Event end timestamp

Table 5.1: Special temporal attributes

PUB_ID

The first attribute added by our extension is the *PUB_ID* attribute, which contains the string representation of a universal unique identifier used to tie start and end

[1]The bag type is used rather than a string in order to allow for bag operators on start or end constraints, which must also apply to instantaneous events.

publications together. The value of this attribute is intended to be unique for each event (a start and end publication pair), and is used to retrieve and update information about the event in the data store.

KBN_PUB_TYPE

Along with the identifier generated for each temporal event, publications include a special attribute which denotes which time marker the event represents, the start or end time, or if the event is instantaneous. The type of the publication is denoted with the *KBN_PUB_TYPE* attribute, which can take one of three values: `START`, `END`, or `INSTANT`. The *KBN_PUB_TYPE* attribute is also linked to the *KBN_START_TIME* and *KBN_END_TIME* attributes described below. Conventionally, any publication in which *KBN_PUB_TYPE* is set to `START` is expected to also include a value for the *KBN_START_TIME* attribute, and the same is true for the value `END` and the *KBN_END_TIME* attribute. Similarly, a value of `INSTANT` expects both a start time and an end time. Also, for each set of publications which share the same value for *PUB_ID*, our implementation assumes that there will only be one publication with a *KBN_PUB_TYPE* of `START` and only one publication with a *KBN_PUB_TYPE* of `END`, or one publication with a *KBN_PUB_TYPE* of `INSTANT`. However, the implementation described here does not enforce these constraints, and does not verify that the conventions have been followed.

KBN_START_TIME and *KBN_END_TIME*

The final two additional attributes used by the temporal extension are *KBN_START_TIME* and *KBN_END_TIME*, which store the timestamps for the start and end of an event, respectively. These two attributes are used as the reference attributes for the $AFTER$ and $DURING$ operators described in Section 4.3.

5.2 Data store implementation

For our implementation, we decided to implement the data store component using two commonly-used relational database systems, Oracle and MySQL. We selected these database systems due to their widespread use, their efficiency in storing linked data (such as attributes being linked to a publication), and as a way to explore some of the more advanced features (including some semantic functions) included in the latest releases of the Oracle 11g database system. This section will first describe the schema we implemented for these relational database systems, then will describe our data store implementations in general before detailing the individual implementations we created.

5.2.1 Schema

The implementation described here uses a very simple schema for storing publications in the database, consisting of only two tables–one for storing publications and one for storing publication attributes. The schema is depicted in Figure 5.1. The

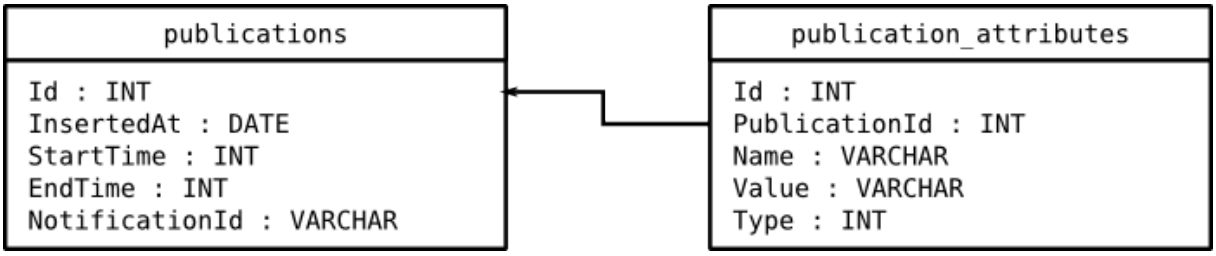

Figure 5.1: Database schema

main table, `publications`, stores some basic information about publications and contains five columns:

- `Id`, the auto-incremented primary key;
- `InsertedAt`, the date and time the entry was created;
- `StartTime`, the start time for the event (the value of the *KBN_START_TIME* attribute);

- `EndTime`, the end time for the event (the value of the *KBN_END_TIME* attribute); and
- `NotificationId`, the universal unique identifier for the event (the value of the *PUB_ID* attribute.

The secondary table, `publication_attributes`, stores information about the attributes of the publications. It also contains five columns:

- `Id`, the auto-incremented primary key;
- `PublicationId`, the foreign key into the `publications` table, based on `publications.Id`;
- `Name`, the attribute name;
- `Value`, the attribute value; and
- `Type`, an integer code representing the attribute type, such as a code of 0 for `null` attribute values, a 1 for Strings, etc.

5.2.2 Data store component overview

We implemented the data store component in our Java implementation in such a way as to follow the abstract factory pattern [32]. The component uses an implementation of a `DataStoreFactory` interface to create a corresponding implementation of the `DataStore` interface, as shown in Figure 5.2. This section describes these interfaces, as well as a generic data store factory implemented to make the configuration of the data stores easier.

`DataStoreFactory` interface design

The `DataStoreFactory` interface provides an implementation-agnostic way for a client of the data store component to get a reference to a `DataStore` object. In the case of this project, the data store object returned is assumed to require some degree of configuration, which may be time consuming. Rather than create a new object

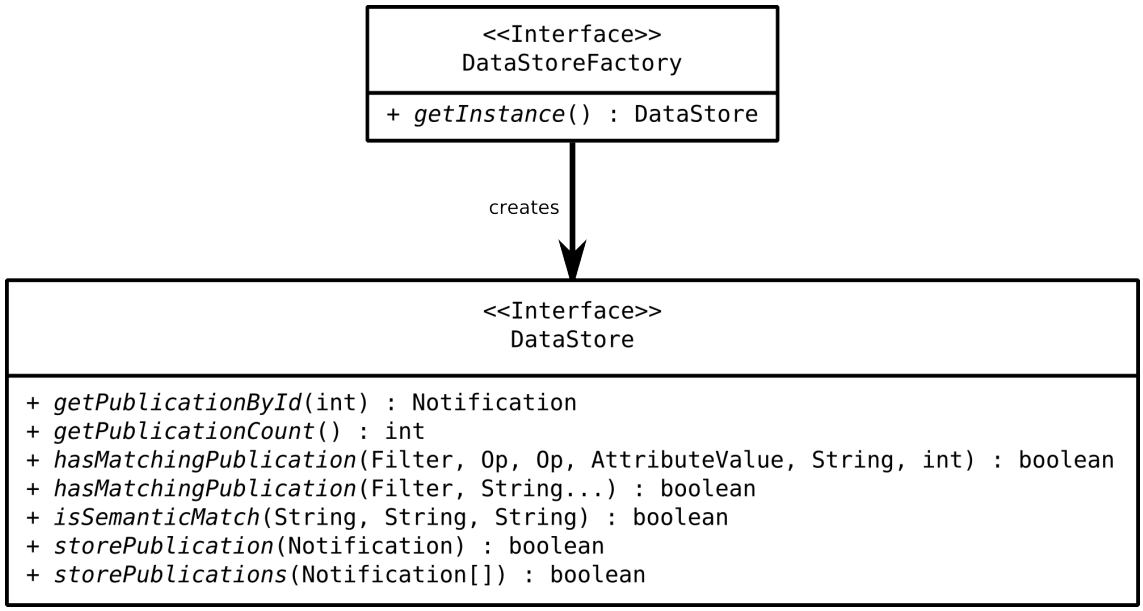

Figure 5.2: `DataStore` and `DataStoreFactory` interfaces

each time a `DataStore` is requested, the data store factory assumes that each of the implementing classes is designed to follow the Singleton pattern [32], although this is not strictly necessary. By using the Singleton pattern, the configuration overhead for each data store implementation can be incurred only once, with `getInstance()` returning a reference to the singleton rather than creating and configuring a new instance for each request.

`DataStore` interface design

The methods which are of particular interest in the data store component are the two `hasMatchingPublication()` methods and `isSemanticMatch()`. The other four methods (`getPublicationById()`, `getPublicationCount()` and the two `storePublication()` methods) simply retrieve a `Notification` object by its id in the data store, retrieve the number of publications currently in the data store, and insert or update stored `Notification`s, respectively.

The two `hasMatchingPublication()` methods both take `Filter`s as arguments and both require the data store implementation to check the passed `Filter` against the stored publications in order to determine whether the `Filter` covers the pub-

lication. This operation could be implemented in a number of ways depending on the capabilities of the underlying data store, and so could potentially benefit a great deal from optimization in the implementation. One of the two method variants requests an additional operator, sub-operator, attribute value, and attribute name, and will only return true if one of the matching publications also matches the has an attribute of the specified name which stands in the relation defined by the given operator to the passed attribute value. The other variant takes a list of one or more conditions which must also be met by the publication. These conditions are expected to be SQL snippets which could be executed within the WHERE clause of a standard SQL query.

The final method, `isSemanticMatch()`, can be used by semantically enabled data stores to simplify the application of the semantic operators added to the KBN by Keeney, et al. [48] If the data store component used by an implementation of the interface does not support semantic operations, this method can also be implemented as a wrapper for some other semantic library which can be invoked from the broker process.

Generic data store factory

The implementation also includes a utility class for configuring and creating instances of the `DataStore` implementation, the `GenericDataStoreFactory`. This class implements the `DataStoreFactory` interface described in Section 5.2.2 and also includes functions for reading a configuration file and registering multiple data store factories. Factories are registered using the static `registerFactory()` method, and can be removed using the `removeFactory()` method. The `hasDefaultFactory()` method checks to see if a default factory has been registered for the zero-argument call to `getInstance()`. If not, the parametrized version of the function must be used to specify a factory instead. The class diagram for the generic data store factory class is shown in Figure 5.3. The generic factory instance reads the configuration from a properties file (datastore.properties in our implementation) which contains the connection details for each listed type of data store factory. An example snippet

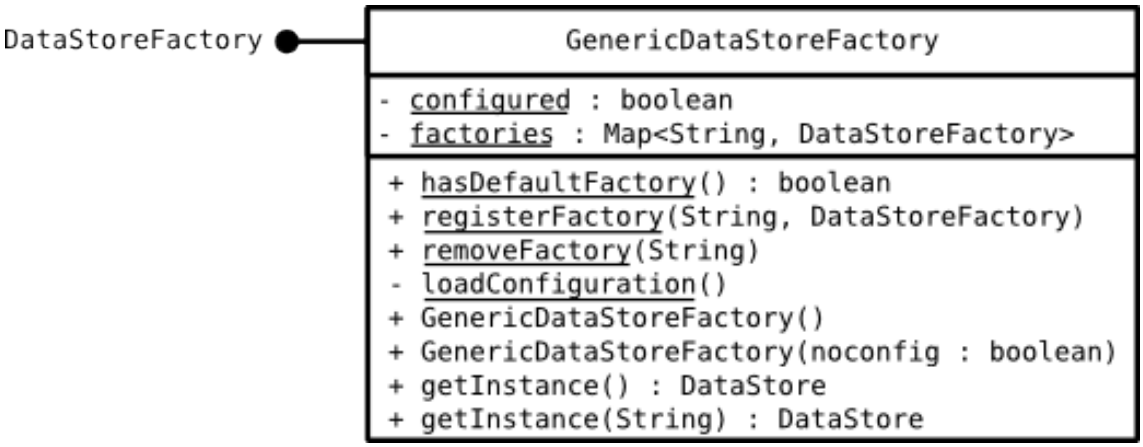

Figure 5.3: `GenericDataStoreFactory` class diagram

of the configuration file is shown in Figure 5.4.

```
datastore.factories=oracle,mysql
datastore.default=oracle

# configuration info for the oracle datastore
oracle.factory=ie.tcd.cs.kdeg.extsiena.database.oracle.OracleDataStoreFactory
oracle.host=localhost
oracle.port=1521
oracle.database=temporalkbn
oracle.username=kbnuser
oracle.password=kbnpass

# configuration info for the mysql datastore
mysql.factory=ie.tcd.cs.kdeg.extsiena.database.mysql.MySQLDataStoreFactory
mysql.host=localhost
mysql.port=3306
mysql.database=temporalkbn
mysql.username=kbnuser
mysql.password=kbnpass
```

Figure 5.4: Example data store configuration

The `GenericDataStoreFactory` first reads the `datastore.factories` property, which lists the names of the individual factories which are to be registered in the system. These names must correspond to further `<name>.factory` entry, which lists the fully-qualified name of the factory class itself. This class must implement

`DataStoreFactory`. A new instance of this factory is created and stored in a map keyed by the name and is used by the system to create new data store instances when that named factory is requested from the generic factory. The `datastore.default` property sets the default factory to be used if no name is provided to the generic factory when requesting a new data store instance.

5.2.3 MySQL data store

Our implementation includes a data store for use with a MySQL 5.0 database, `MySQLDataStore`. This implementation uses the schema described in Section 5.2.1 and the abstract factory pattern described in Section 5.2.2, consisting of two classes: `MySQLDataStore` and `MySQLDataStoreFactory`. This implementation takes a very naïve approach to finding matching events from the database, using simple SQL queries to find any events which match the temporal constraints imposed by the operators and looping through them in code to test against the filter. As such, the MySQL implementation is expected to scale linearly with the size of the result set.

In order to ensure that our implementation does not lose too much performance in connecting to the database, we have implemented connection pooling for the actual database connections using the Apache Common DBCP component [30], which provides Java libraries for database connection pooling using the functionality of the JDBC 2.0 drivers provided by database vendors. The actual connections to the database are handled using the Apache `BasicDataSource` class, which uses the underlying driver to make connections to the database which are then used multiple times as more connections are requested or connections are closed.

As another note, the MySQL 5.0 series of database servers provide no semantic support, so the `isSemanticMatch()` of this implementation simply returns a result of `false`. When using this data store, the semantic operators must be handled using some other tool, such as Jena [41], which is the library used by the KBN in our implementation.

5.2.4 Advanced Oracle data store

Along with the MySQL data store, our implementation uses two Oracle data store implementations, one which uses many of the advanced features of the Oracle 11g data store and the other of which uses only the standard RDBMS features. This section describes the more advanced implementation, including the Oracle features it uses. These features include the ontology-extended relational queries introduced as part of Oracle's Semantic Technologies [58] and the ability to invoke static Java methods as stored procedures from the Oracle 11g instance [20].

The core of the advanced data store implementation are the classes which implement the interfaces described in Section 5.2.2: `OracleDataStore` and `OracleDataStoreFactory`. The database connection used by `OracleDataStore` utilizes the built-in connection pooling of the Oracle driver (provided by the `OracleConnectionCache` framework) which is included in the `OracleDataSource` provided by the JDBC 2.0 driver. This allows the system to reuse existing connections to the database each time the broker connects rather than creating and configuring a new connection. The Oracle driver handles all the connection caching and reaping automatically. Also, the Oracle implementations require the addition of two sequence objects in the database, `publication_seq` and `publication_attribute_seq`, in order to handle the automatic numbering for the database entry identifiers.

In order to perform the matching of stored events, the advanced Oracle data store invokes a stored procedure, `MATCH()`, from the Oracle database. This stored procedure is actually a static Java function from the `OracleDataStoreMatcher` class, which parses a filter from a passed string and applies it to a publication retrieved from an internal data store (the stored procedure is in more detail in Section 5.2.4). Any publication which matches the filter is then included in the result set and returned to the data store class. As this is limiting the size of the result set and there is no external looping required to analyze these results, we expect that this implementation would perform rather better than the simpler MySQL and Oracle

implementations, perhaps growing with the log of the result set size rather than linearly.

Finally, the Oracle 11g instance includes support for semantically-enhanced queries comparing against a loaded ontology. This allows the Oracle data store to offer the `isSemanticMatch()` function by the `SEM_RELATED` function to determine if a particular triple is included in a loaded ontology. The semantic technologies included in the Oracle 11g installation are described in more detail in a later subsection.

Java stored procedures

In addition to the standard RDBMS features, the Oracle database comes bundled with its own Java virtual machine using the included `loadjava` utility, which allows Java objects to be loaded into the database as resources which can be called later, which is not possible with other available database solutions. These resources can either be Java source files or code, which is compiled and resolved against the other loaded libraries, or Java class files, which are only resolved against the previously loaded resources. Any method to be invoked as a stored procedure must be a static method, but can otherwise make use of any resources that are accessible to the database user.

In the Oracle data store implementation, the `OracleDataStoreMatcher` class acts as the container for the stored procedure, `match()`, which is invoked during SELECT queries used by the temporal and filter operators. The `match()` function accepts the string serialization of a filter (as marshalled by Siena's communication utilities) and the database id of the publication currently being examined by the query, returning `1` if the publication is a match and `0` if not.

Performing these operations within the database makes the logic of the data store much simpler, as the query only returns rows which are already checked against the filter, but it adds some other technical complications. For one, the Oracle virtual machine is run from a different process than the rest of the KBN, so none of the

objects or configuration of the KBN can be accessed directly by the matching class. Also, the entire codebase of the KBN must be loaded by the database, including all of the classes invoked in the course of matching filters. This could potentially lead to some extra overhead for each call because the matcher must reconfigure itself each time.

Internal Oracle data store

In order for the stored procedure to match nested temporal or filter operators (as well as the ontological operators), the matching procedure itself may sometimes need to invoke the data store to test for matching publications. This is complicated somewhat by the restrictions placed on stored procedures by the Oracle VM, which do not allow connections of the sort attempted by the standard Oracle JDBC driver. However, the Oracle VM also includes a special implementation of the JDBC driver which can be used from within the database when a connection is needed from the Java stored procedure.

The extension also includes a special implementation of `DataStore` and `DataStoreFactory` which invoke this driver from within the matching function, called `InternalOracleDataStore` (which extends `OracleDataStore`, itself an implementation of `DataStore`) and `InternalOracleDataStoreFactory` respectively. The data store factory is registered in the `GenericDataStoreFactory` as the default for the Oracle VM environment with each invocation of the match function, so that any future requests for a data store instance will use the internal data store rather than attempting to connect to the database with the standard JDBC driver.

Oracle semantic technologies

Oracle 11g database systems include support for storing and querying semantic data, including using ontological information to extend standard relational queries. This is especially useful for storing and querying data which can be related semantically as well as by value. For instance, a standard relational query for "EnginePart" would not return an entry for a "SparkPlug", simply because the values do not match.

However, if the two entries are connected ontologically such that a spark plug is listed as an engine part, then the semantically-enhanced relational queries would find the relevant data, allowing for more flexibility in the way data is represented in the database. Our implementation utilizes this feature with the `isSemanticMatch()` function, which can be used in place of the Jena [41] Java reasoner currently used by the KBN implementation.

The semantic utilities provided in the database are included as part of the `SEM_APIS` package of PL/SQL subprograms introduced as part spatial data features of the Oracle 10g database. Semantic data is first loaded as triples into the database, which generates an `SDO_RDF_TRIPLE_S` type object to represent it within the Oracle relational data. The actual data for these triples is stored as metadata in the `MDSYS.RDF_VALUE$` table, which is automatically maintained by the database itself. Triples can be inserted into the database in bulk via a staging table and PL/SQL script, using a provided Java client interface to load an N-Triple format file, or individually using the provided `SDO_RDF_TRIPLE_S` constructor in an insert query.

Once loaded, the triples can then be used to construct a semantic model, which models the triples as a directed graph where the subject and object are stored as nodes connected by the predicate, stored as a directed edge between the nodes. The models in the Oracle database are also automatically maintained as metadata in the `MDSYS.SEM_MODEL$` view and are only accessed via the provided PL/SQL subprograms. These models can then be used to make queries on the semantic data.

In contrast to Jena, the Oracle database need only load the ontological model once, because once the model has been stored in the database and the appropriate indices have been created, the model can continue to be queried. Jena, on the other hand, must have it's model reloaded every time a new KBN broker is started, adding time to the start-up of the broker. However, Jena does provide greater flexibility in the type of reasoners allowed and in the particular rules used for reasoning, as the Oracle semantic implementation only allows a small number of reasoners culminating

in the OWLPrime[1] rulebase which is the largest included set of rules for the Oracle reasoner. These can be extended with custom rulebases, however, but the process is more complicated than the process is with Jena.

The implementation of the Oracle data source uses the included `SEM_RELATED` operator, which retrieves rows from the database based on their semantic relatedness within a specified set of semantic models with respect to a specified set of ontological rulebases, such as RDFS or the Oracle-created OWLPrime. Using the specified models and rulebases, `SEM_RELATED` checks if the passed subject and object are linked via the passed predicate. This means that the matching of the semantic operators is performed within the database itself rather than invoking the Jena reasoner. Using this feature, the Jena reasoner could be completely eliminated from the KBN brokers, decreasing the number of library dependencies for the KBN Java codebase.

5.2.5 Simple Oracle data store

Along with the advanced Oracle data store, we also implement another data store which uses the same Oracle database without invoking the Java stored procedures. Rather, the `OracleJDBCDataStore` implementation checks for matching events in the data store by retrieving all the rows and iterating over them in code, similar to the MySQL implementation described in Section 5.2.3. This implementation can also benefit from the Oracle semantic technologies as well, although it is not necessary.

5.3 Operator implementation

In our implementation, the three operators described in Section 4.3–$AFTER$, $DURING$, and the $FILTER$ operator–all implement a common interface, `ComplexOperator`, shown in Figure 5.5.

[1]OWLPrime supports all RDFS operations and a subset of OWL capabilities, including many of the basic comparisons of classes, properties, and individuals but excluding many set relations such as cardinality, unions, intersections, and enumeration.

```
<<Interface>>
ComplexOperator
+ apply(String, AttributeValue, Filter) : boolean
+ same_op(ComplexOperator) : boolean
```

Figure 5.5: The `ComplexOperator` interface

This interface includes two methods, `apply()` and `same_op()`. The `apply()` method takes three parameters: the *name* of the reference attribute to which the attribute is applied, the *value* of the reference attribute (as an `AttributeValue`), and the target filter to which the operator is to be applied. If the operator does apply, the method returns `true`, and it `false` otherwise. The `same_op()` method checks to see if the passed *ComplexOperator* describes the exact relation described by the invoked operator.

The interface is directly implemented by two classes, `FilterOperator` (which implements the $FILTER$ operator from Section 4.3) and the abstract class `TemporalOperator`. The `TemporalOperator` class is then further extended by the implementations of $AFTER$ and $DURING$, the `AfterOperator` and the `DuringOperator` respectively.

5.3.1 `FilterOperator`

The `FilterOperator` class implements the `ComplexOperator` interface, as shown in Figure 5.6.

The class has three members:

- `limit`, which stores the maximum number of results to be examined when applying the operator;
- `operator`, which represents the main KBN operator to be applied to matching publications; and

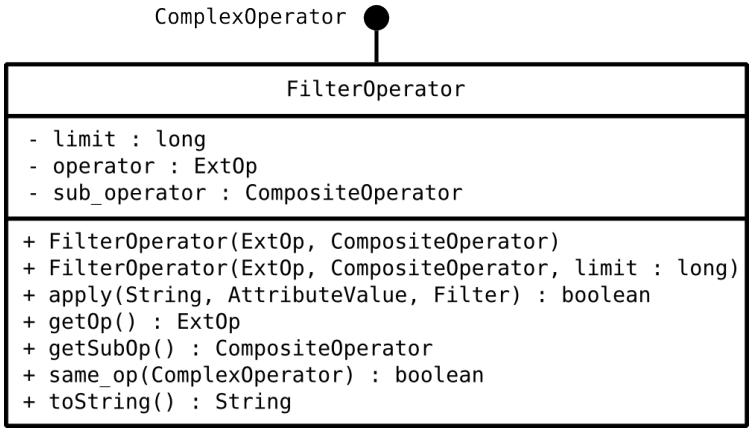

Figure 5.6: The `FilterOperator` class

- `sub_operator`, which stores the secondary operator to be used if (and only if) `operator` is a bag operator.

The two constructors generate a new instance of the `FilterOperator`, either with a passed limit or using the default limit set in the KBN code (currently 10000). The `apply()` method retrieves a reference to a data store instance from the `GenericDataStoreFactory` and invokes `hasMatchingPublication()` with the stored operators and limit and the passed attribute value and target filter. The `same_op()` method returns true if the passed operator is a `FilterOperator` and if the operator, sub-operator, and limit are the same. An example of creating a `FilterOperator` is given in Figure 5.7.

5.3.2 `TemporalOperator`

The other class to implement `ComplexOperator` is the abstract `TemporalOperator` class, along with its two sub-classes, `AfterOperator` and `DuringOperator`. These three classes are shown in Figure 5.8.

The `TemporalOperator` class itself implements only the `apply()` method, which retrieves a `DataStore` from `GenericDataStoreFactory` and invokes `hasMatchingPublication()` using the two conditions returned by the abstract condition methods of the class along with a condition which prevents an publication from matching with its own event. All of the other methods listed in the

```
// Example of using the FilterOperator
public void useFilterOperator(Siena kbn, Notifiable subscriber) {
    // set the main operator (less than)
    ExtOp main = ExtOp.LT;
    // set the sub operator (null here, since we're not using a bag)
    CompositeOperator sub = null;
    // set the limit
    long limit = 500;
    // create the operator
    FilterOperator filterop = new FilterOperator(main, sub, limit)
    // set the reference attribute name
    String reference = "price";
    // create a target filter
    Filter target = new Filter();
    target.addAttributeConstraint("attr", "somevalue");
    // create the outer filter
    Filter outside = new Filter();
    outside.addAttributeConstraint(reference, new AttributeConstraint(filterop, target));
    // subscribe
    try {
        kbn.subscribe(outside, subscriber);
    } catch (SienaException e) { }
}
```

Figure 5.7: Example of using `FilterOperator`

`TemporalOperator` class are implemented in the subclasses, `AfterOperator` and `DuringOperator`, which implement the $AFTER$ and $DURING$ operators respectively. Some example code to use these operators is shown in Figure 5.9.

`AfterOperator`

The `AfterOperator` class implements the $AFTER$ operator described in Section 4.3. It has one member variable, `limit`, which stores the maximum number of time steps into the past that the operator will consider when testing for a match. This field is filled from the constructor when the operator is used.

The two conditions specified by the `AfterOperator` (with `getConditionOne()` and `getConditionTwo()`) both relate to the end time of the stored events the operator is applied against:

1. The end time, Y_E, of the stored event Y must be less than the value of the reference attribute, X_R (i.e., $Y_E < X_R$) and

2. The end time must be greater than or equal to the value of the reference attribute minus the limit, L (i.e., $Y_E \geq (X_R - L)$).

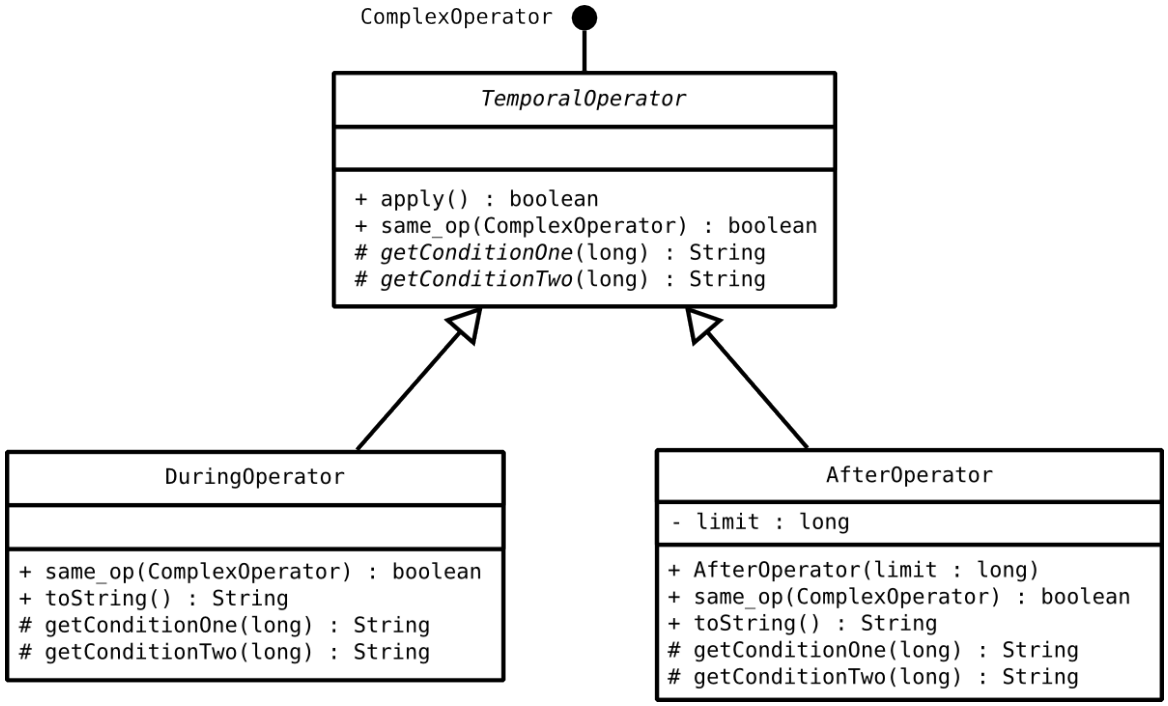

Figure 5.8: `TemporalOperator` and its subclasses

`DuringOperator`

The `DuringOperator`, which implements $DURING$, has no member variables, as there is no additional parameterization to be done for this operator. The conditions it specifies relate to both the start and end times of the stored events:

1. The start time, Y_S of the stored event must be less than or equal to the value of the reference attribute (i.e., $Y_S \leq X_R$) and

2. The end time of the stored event must either be greater than or equal to the value of the reference attribute or it must be `null` (i.e., $Y_E \geq X_R \vee Y_E$ `is null`).

5.4 Utility classes

In addition to the data store and the temporal operators, we implement a number of utility classes which encapsulate the functionality and conventions added by the new component and operators. This section will describe those new classes.

```
// Example of using the temporal operators
public void useTemporalOperators(Siena kbn, Notifiable subscriber) {
    // create a target filter
    Filter target = new Filter();
    target.addAttributeConstraint("attr", "somevalue");
    // create an outer filter
    TemporalFilter overlaps = new TemporalFilter();

    // create the during operator
    DuringOperator duringop = new DuringOperator();
    // add a start constraint to use the during operator
    overlaps.addStartConstraint(new AttributeConstraint(duringop, target));

    // set the time limit for the after operator
    long limit = 50;
    // create the after operator
    AfterOperator afterop = new AfterOperator(limit);
    // add an end constraint to use the after operator
    overlaps.addEndConstraint(new AttributeConstraint(afterop, target));

    // subscribe
    try {
        kbn.subscribe(overlaps, subscriber);
    } catch (SienaException e) { }
}
```

Figure 5.9: Example of using the `TemporalOperators`

`TemporalCovering`

In order for our implementation to correctly function within the Siena/KBN framework, we need to address the *covering* relations employed by Siena, as described in Section 3.2. The `TemporalCovering` class performs this task with its `covers()` method, which takes two `AttributeConstraint` objects as parameters and determines whether or not one covers the other.

For the operators in our extension, the covering relationships depend heavily on the internal target filters used in the operators, and cannot be determined otherwise. Due to this constraint, the `TemporalCovering` class only returns a result of `true` if the two attribute constraints contain the same `ComplexOperator` (using the operators' `same_op()` method) and the internal filter of the left-hand operator covers that of the right-hand operator.

TemporalFilter

To simplify the use of the new start and end time semantics which represent events in our extension, we have added a small extension to the Siena `Filter` class to help with adding constraints to the start and end times. This extension, `TemporalFilter`, simply adds two new methods, `addEndConstraint()` and `addStartConstraint()`, which automatically add the appropriate constraints to match the conventional values used in the temporally-enhanced publications. The class diagram showing the new methods is shown in Figure 5.10.

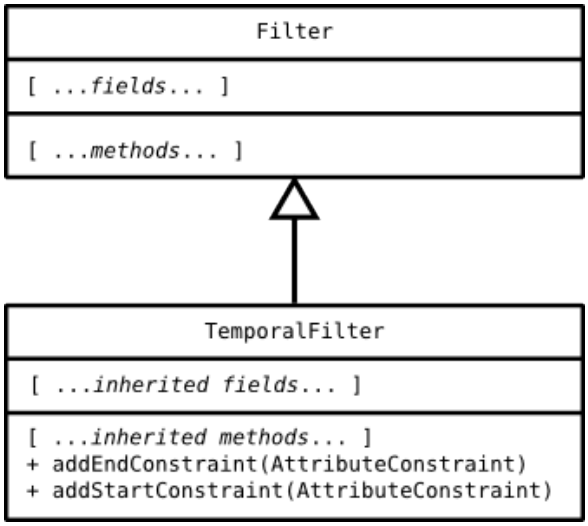

Figure 5.10: `TemporalFilter` class diagram

TemporalNotification

Similarly, we also introduce a new extension to the Siena `Notification` class, `TemporalNotification`. This class simplifies the addition of the start and end timestamps to publications which take advantage of the new temporal extension to the KBN. The two new methods, `setEnd()` and `setStart()`, create and set all the attributes conventionally used by the temporal notification framework. The class diagram for the `TemporalNotification` class is shown in Figure 5.11.

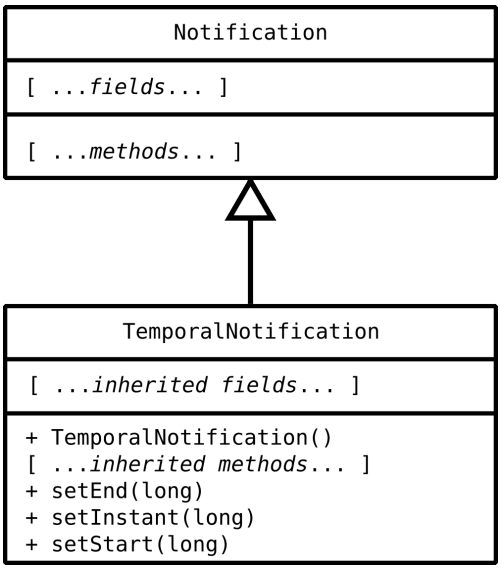

Figure 5.11: `TemporalNotification` class diagram

5.5 Siena/KBN modifications

The features described in this chapter also require some modifications to the existing Siena/KBN framework. This section describes those modifications.

`HierarchicalDispatcher`

The `HierarchicalDispatcher` class in the KBN is the main class for each broker, as it implements the `Siena` interface. The only change necessary for this extension is to add a `store` class member, which stores a reference to the default `DataStore` instance returned by the `GenericDataStoreFactory`, which is invoked when the dispatcher initializes. Also, it is necessary to specify whether the broker will be using the semantic features of the database or an external reasoner in the dispatcher initialization.

The final change to the `HierarchicalDispatcher` relates to the subscription process. In order for the new operators to successfully retain all relevant publications, the first-level (non-nested) inner filters stored by each of the new operators must be sent up the subscription tree as well. Our implementation alters the `subscribe()` method so that it extracts any first-level inner filters and creates a new subscription

by the broker for those filters as well.

AttributeConstraint

Both of the new types of operator added in this extension require a Siena `Filter` object as an operand, but the existing `AttributeConstraint` class does not allow for `Filter`s to be given as values. Furthermore, since the new operators do not extend the old operator classes, there is no way of storing those operators or testing for their presence in the existing design. In order to allow the new operators to work within the Siena framework, the `AttributeConstraint` class must be extended with new fields and new functions which store and access the new operators and the `Filter` target values.

SENP modifications

The new operators also require a change to Siena's SENP (Simple Event Notification Protocol) in order for the new operators to be marshalled and transmitted by the brokers. These changes are of particular note, because they will prevent brokers running previous versions of Siena or the KBN from recognizing subscriptions using the new operators.

In Siena, messages are marshalled and transmitted as specifically formatted strings which are parsed by the `SENPBuffer` class. These strings include special characters or character combinations to represent the Siena operators, such as ">" to represent the greater-than operator and "`*`" to represent the substring operator. These operators were extended by the KBN, introducing such combinations as "`@=`" for ontological equivalence and "`#<`" for the subbag operator. Similarly, the new operators added to the KBN in this project were added to the `SENPBuffer` class (and to the SENP itself) as described in Table 5.2.

Correspondingly, we modified the SENP writer, `SENPWBuffer`, to encode the new operators when encoding the `AttributeConstraint`s that contain them. Each of

Symbol	Next token	Operator
`$=`	<*filter*>	During operator
`$>`	<*limit*><*filter*>	After operator
`^`	<*operator*><*filter*>	Filter operator

Table 5.2: Temporal and filter operator codes

the new operators includes a `toString()` function which prints the appropriate symbol to a string, along with the following tokens (the limits the operators) required by the `AfterOperator` and the `FilterOperator`. The `SENPWBuffer` then encodes the target filter as it would normally encode a filter, to be read by the `SENPBuffer` on the receiving end.

5.6 Summary

In summary, we implement the following components based on the design described in Chapter 4 and modify the existing KBN codebase to use them:

- A simple database schema for storing publication data in a relational database;
- A generic data store factory for use in configuring the data store;
- Two naïve data store implementations using the JDBC drivers for MySQL and Oracle;
- A more advanced Oracle data store implementation utilizing Java stored procedures and Oracle's semantic technologies;
- Two temporal operators, `AfterOperator` and `DuringOperator`, which implement the `ComplexOperator` interface;
- A generic `FilterOperator` which also implements `ComplexOperator`;
- Two utility classes, `TemporalFilter` and `TemporalNotification`, which ease the use of the new components; and
- `TemporalCovering`, which implements the covering relations used by Siena for the new operators.

The modifications include some changes to the `HierarchicalDispatcher` class to attach the data store and route inner filters up the tree as subscriptions and to the `AttributeConstraint` class in order to make it support the new operators and take filters as target values. We also added the new operators to Siena's transfer protocol, SENP.

The next chapter explains the tests used to evaluate the implementation in terms of the scalability of the data store implementations, including some analysis of that evaluation.

Chapter 6

Evaluation

The previous chapter detailed the implementation of the design we proposed in Chapter 4, including the various data store implementations we implemented for the system. This chapter will describe the evaluation framework we used to test the scalability of our implementation. The first section will briefly describe the evaluation framework, followed by a description of the tests run on the empty databases with their results, and finally the tests run with pre-seeded data already in the data stores.

6.1 Framework

The tests described in this section were all run on a Dell Server PE1900 with an Intel Xeon (Quad-core) 2.33 GHz processor, 4 GB of RAM, and 585 GB of hard disk space running Microsoft Windows Server 2003 R2 Standard Edition with Service Pack 2. The Java implementation was compiled with a Java 2 Standard Edition runtime environment, version 1.5.0_11, and was executed with a Java SE runtime version 1.6.0_15. The Oracle data store implementations both used an Oracle Database 11g 11.1.0.7.0 installation using the official Java 1.6 JDBC driver, and the MySQL data store implementation connected to a MySQL 5.1.28 installation (on a community license) using the MySQL JDBC connector version 5.1.8. All of the semantic operators were run using a Pellet-reasoned version of the W3C Wine Ontology [80]. The tests use KBN Version 4 (Version 3 plus the new temporal extensions described in

Chapter 5).

Statistics were collected using a `Statistics` class included as part of the implementation. This class collected data on the processing time taken to complete various tasks with the KBN broker in order to see if how the system scaled with the number of stored events. We collected the following statistics from the broker as each publication was received:

- The id of the publication (in order to group the results),
- The number of events stored in the data store at the time the publication was received,
- The number of subscribers to which the publication was delivered,
- The time (in milliseconds) taken to insert the event in the database (for *start* publications),
- The time (in milliseconds) taken to update the corresponding event in the database (for *end* publications),
- The average time (in milliseconds) to successfully apply each filter operator checked against the publication,
- The average time (in milliseconds) to unsuccessfully apply each filter operator against the publication,
- The average time (in milliseconds) to successfully apply each temporal operator to the publication,
- The average time (in milliseconds) to unsuccessfully apply each temporal operator to the publication, and
- The total time (in milliseconds) taken to process the publication.

6.2 Test 1: Empty data store

The first series of tests attempted to establish how the processing time for the extension grew with respect to the number of stored publications, starting with an empty data store. For this test, we connected an example publisher and an example subscriber to a single temporally-extended KBN broker running on the server described in Section 6.1. The example subscriber generated fifty unique subscriber instances, each subscribing with one of the subscriptions shown in Table 6.1, randomly-selected with a uniform description (a filter used as a target value is denoted by F_X where X is the filter name).

Filter	Attribute	Operator	Target
A	`A`	$<$	8
B	`B`	$ONTPROP_{\texttt{<wine:hasColor>}}$	`<wine:Red>`
C	`A`	$FILTER_{<}(200)$	F_B
D	*End*	$DURING$	F_A
E	*Start*	$AFTER(500)$	F_A
F	*Start*	$AFTER(125)$	F_A
X[a]	`C`	$>$	10
G	`A`	$FILTER_{<}(200)$	F_X

[a] This filter is only used as the internal filter for the following filter operator.

Table 6.1: Subscriptions for Test 1

The example publisher connected to the same instance and published information about 250 events, with one beginning every 10 seconds (in order to allow the broker to fully process the request) and lasting between three and eight seconds (in order to test varying event durations), at which time the end event for that event was published. The publications contained two attributes, `A` and `B`, which were each assigned an independent random value. `A` was set to an integer value between one and ten, inclusive, and `B` was set to the URL of one of ten wines from the Wine Ontology, five red and five white.

This test was run fifty times for each of the three data store implementations discussed in Chapter 5: the advanced `OracleDataStore`, the simple `OracleJDBCDataStore`,

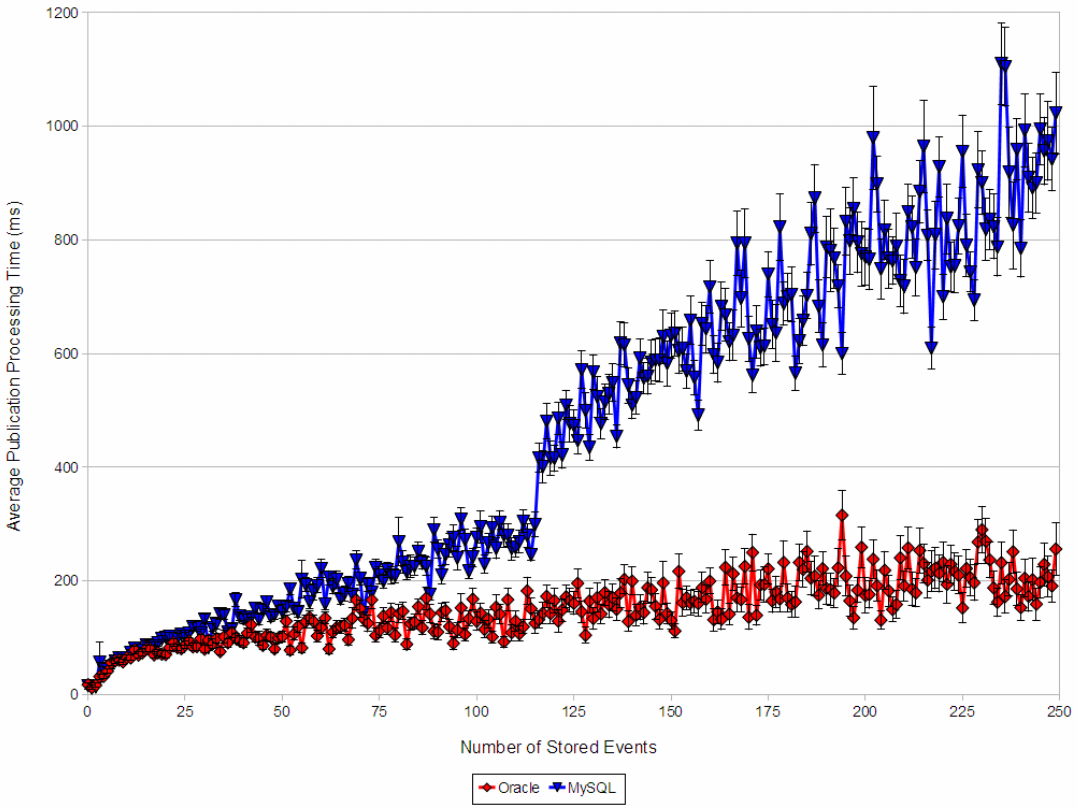

Figure 6.1: Average processing time of the `OracleJDBCDataStore` and the `MySQLDataStore` implementations

and the `MySQLDataStore`. Based on the algorithms employed, we predicted that the processing time would scale at most linearly with the number of events stored in the data store. The results of the tests using the `OracleJDBCDataStore` and `MySQLDataStore` implementations are shown in Figure 6.1. As the graph indicates, the average processing time for each publication does appear to increase linearly with the number of events stored in the data store, at least for the MySQL results which match a linear regression with a high degree of confidence ($R = 0.989$). The simple Oracle JDBC implementation also strongly fits a linear regression ($R = 0.854$), but is also a strong fit for a logarithmic regression based on the test data ($R = 0.819$), showing that the processing time for the `OracleJDBCDataStore` may grow with the log of the stored events rather than growing linearly.[1]

[1]The error bars in all the graphs shown in this document represent the standard error, not the standard deviation from the mean.

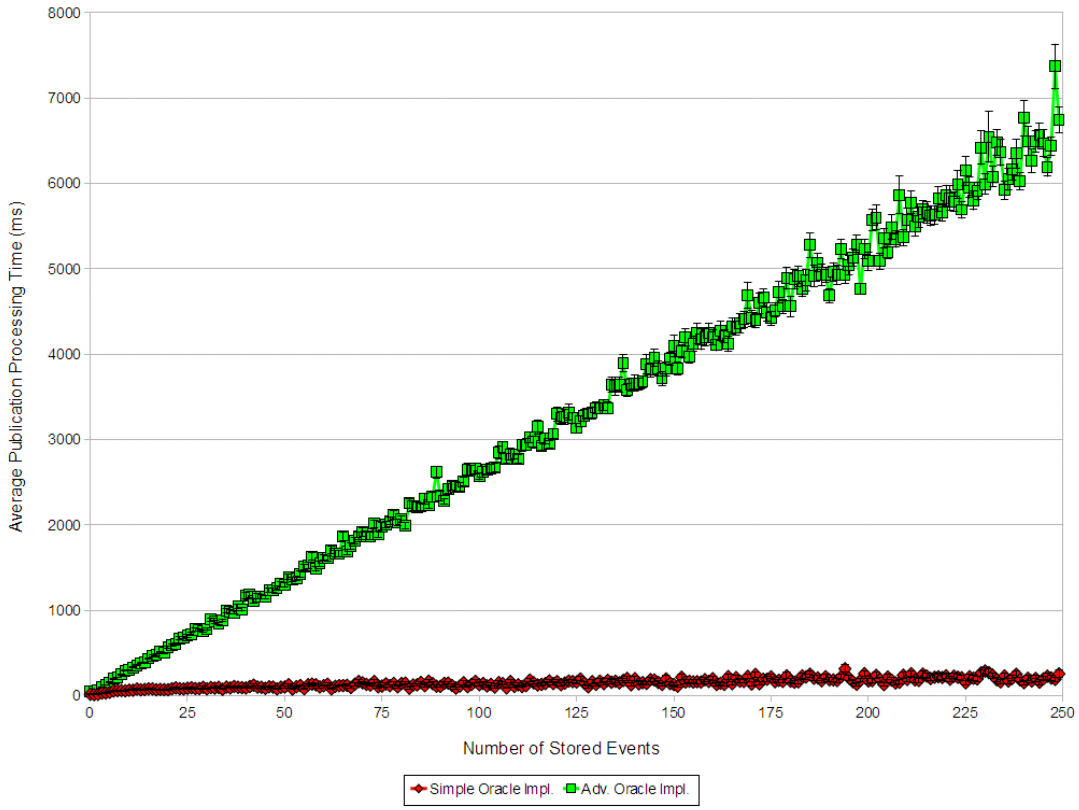

Figure 6.2: Average processing time of the Oracle data store implementations

The more advanced `OracleDataStore` class also displayed a linear increase with the number of stored events ($R = 0.999$), but the increase was dramatically greater than the simpler `OracleJDBCDataStore`, which did not utilize the Java stored procedure invocations used by the more complex version. Figure 6.2 compares the two Oracle data stores directly based on their results from Test 1.

As shown in Figure 6.2, the more advanced Oracle data store may grow linearly, but for only 250 stored events (a *very* small number) the processing time for *each* publication takes an average of nearly seven seconds! This sort of processing time is very troubling, as the results should have been more akin to those of the simpler Oracle JDBC implementation and the MySQL data store implementation shown in Figure 6.1. The increased processing time for each publication may well be due to the overhead involved with the invocation of the Java stored procedure defined in the `OracleDataStoreMatcher` class described in Section 5.2.4. In order to check for

matches, the matcher class must load most of the KBN infrastructure and invokes a very complicated sequence of calls in order to parse the passed filter, retrieve the attributes from the publication being examined, and apply any operators to determine the match. This may require the Oracle 11g instance to dynamically load and unload many Java classes (although the logging and other utilities used by the matcher were loaded only a very small number of times), which could dramatically increase the time taken to perform each query, especially as the number of events stored in the database grows.

Summary: Test 1

The first test published 250 randomly-generated publications to 50 randomly-selected subscribers using a single temporally-extended KBN broker and an initially empty data store. Each of the three implemented data stores (`OracleDataStore`, `MySQLDataStore`, and `OracleJDBCDataStore`) was used for 50 complete runs of the test. The results demonstrated that all three scaled (at most) linearly with the number of stored events, with the simpler `OracleJDBCDataStore` showing the best performance. The poor performance of the more complex `OracleDataStore` was surprising as it did not seem to be affected by subsequent changes in the configurable variables of the test (e.g., delay between publications, number of publications), but it may be explained by the increased overhead necessary to invoke the Java stored procedures used to match the new operators.

6.3 Test 2: Pre-loaded data

The second test set utilized much the same set-up as the first, involving a single temporally-extended KBN broker, a simple subscriber, and a simple publisher. However, the second set of tests operated on a pre-filled database containing 50,000 randomly-generated events (using the same publication attributes as the publisher in the first test; see Section 6.2) and focused specifically on the time taken to process the $FILTER$ operators, which we determined take the most time due to their unconstrained nature (the temporal operators took an average of around 10ms to

process). The fifty subscribers for the second test used one of two filters, randomly-selected from those described in Table 6.2. The publisher used for the second test is the same as the publisher used for the first test, but with a three second delay rather than a ten second delay between publications.[2]

Filter	Attribute	Operator	Target
X[a]	B	$ONTPROP_{\texttt{<wine:hasColor>}}$	`<wine:Red>`
A	A	$FILTER_{<}(\alpha)$[b]	F_X
Y[a]	C	$>$	10
G	A	$FILTER_{<}(\alpha)$[b]	F_X

[a] This filter is only used as the internal filter for the following filter operator.
[b] The limit on the filter operators used in these subscriptions is one of the variables manipulated in this test.

Table 6.2: Subscriptions for Test 2

The second set of tests were each run with varying values for the filter operator limit (200, 500, 1000, 2000, and 5000 results), all using the `OracleJDBCDataStore` implementation. We expected that the average processing time would remain nearly constant, due to the limit imposed on the filter operator result set size. As expected, the average publication processing time for the second test (with a filter operator limit of 200 results) using the simple Oracle data store implementation hovered around 200 milliseconds, as shown in Figure 6.3.

These results show that the processing time scales linearly not in the number of total events stored in the database, but rather that the processing time is a function of the size of the result set returned for each query. We expect that the processing time will increase as the limit on the result set increases. In order to test this prediction, we ran the second test ten times each with filter operator limits of 200, 500, 1000, 2000, and 5000 results. The average publication processing time for each limit (along with the standard deviation and standard error) are shown in Table 6.3 and a graphical depiction of those results is shown in Figure 6.4.

[2]The delay was decreased in order to speed up the testing process and Test 1 showed that the JDBC implementation did not approach the previous 10 second mark.

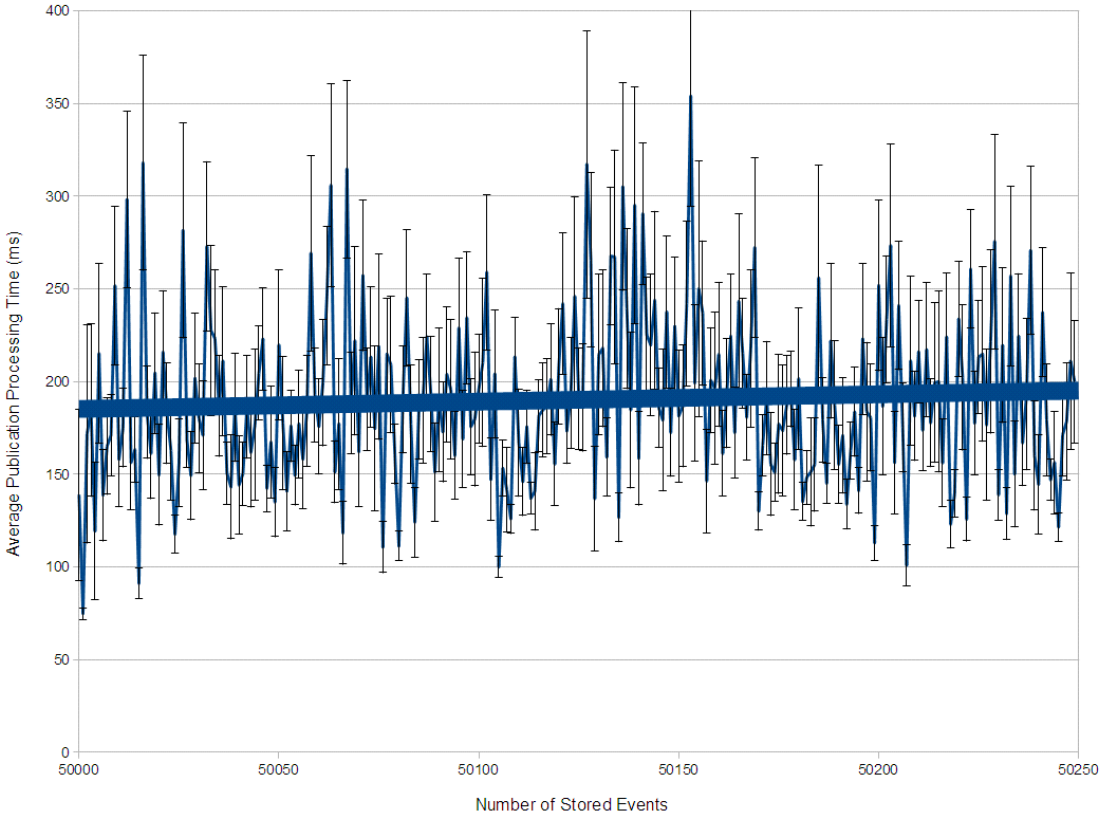

Figure 6.3: Average publication processing time with a pre-filled data store

Limit	Avg. Time (ms)	Std. Dev.	Count	SEM
200	193.20	153.62	4949	2.18
500	284.72	371.35	4954	5.28
1000	444.55	759.29	4949	10.79
2000	753.96	1556.41	4949	22.12
5000	2957.66	5060.09	4947	71.94

Table 6.3: Average processing times based on filter operator limit

As expected, the processing time does increase as the filter operator limit increases, although the increase is not quite as linear as expected. The first four data points (200, 500, 1000, and 2000 result limits) do appear to increase linearly, but the increase up to 5000 results shows a greater marginal increase than expected. This increase could be due to the processing time approaching the delay between publications, leading to a queuing effect as the publications begin to stack up awaiting processing. Siena only processes a single publication at a time at each broker, and

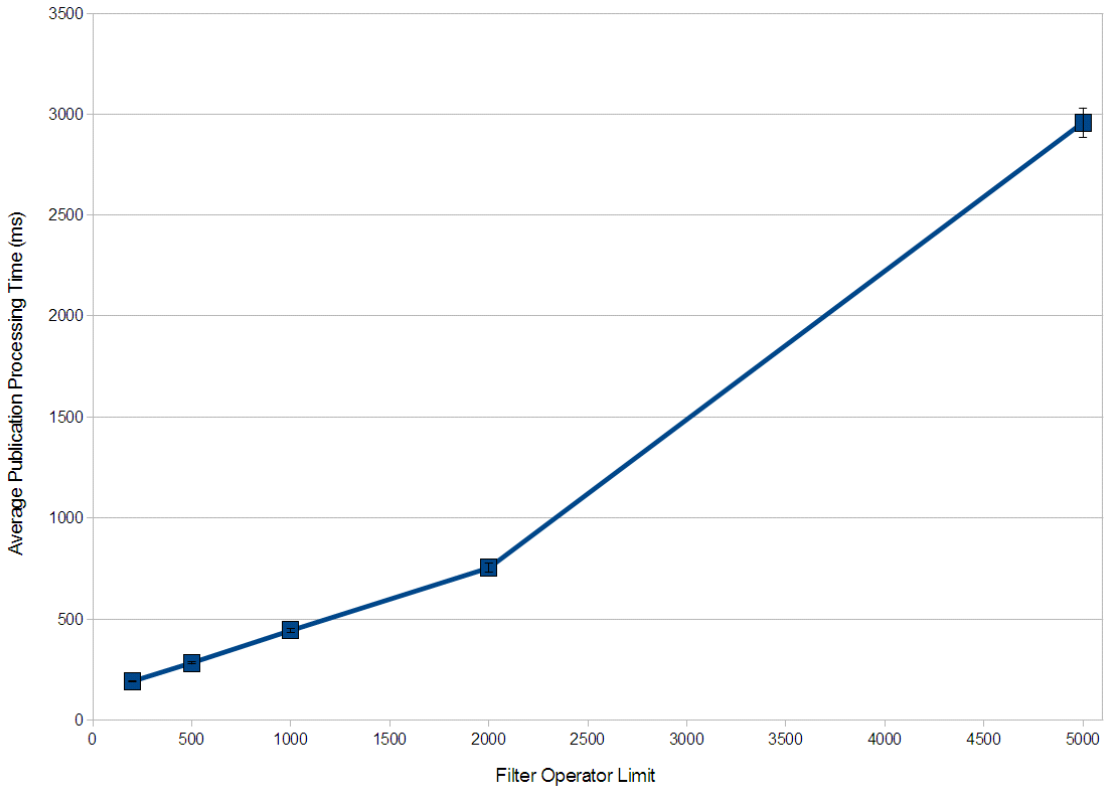

Figure 6.4: Average processing times based on filter operator limit

the statistics gathering framework counts this wait time as part of the total.

In order to see more clearly the effect of the increased limit on the processing of the filter operators, we also gathered statistics on the processing time devoted only to the application of the filter operators for the publications published during the test. This time is also expected to increase linearly with the size of the limit. The average filter operator processing times are shown in Table 6.4 (including the averages for matched and non-matched publications), and the combined average values are plotted as a graph in Figure 6.5.

By this measure, the results of Test 2 do show that the filter operator processing time does increase linearly with the limit on the result set returned by the queries used to evaluate the operators ($R = 0.999$), which agrees with our earlier expectations. Also, the average time taken to process matching filter operators is far less than the average time taken for those publications that do not match (which is the

Limit	Avg. Time (ms)	Std. Dev.	Count	SEM
200	31.05	31.95	4492	0.48
500	37.39	45.15	4506	0.67
1000	37.55	58.55	4447	0.88
2000	45.13	143.58	4460	2.15
5000	83.89	488.41	4433	7.34

(a) Matching publications

Limit	Avg. Time (ms)	Std. Dev.	Count	SEM
200	35.45	56.10	4949	0.80
500	67.79	151.17	4954	2.15
1000	116.82	281.90	4949	4.01
2000	256.52	690.74	4949	9.82
5000	587.88	1573.70	4947	22.37

(b) Non-matching publications

Limit	Avg. Time (ms)	Std. Dev.	Count	SEM
200	40.68	55.51	4949	0.79
500	73.67	148.61	4954	2.11
1000	118.80	275.42	4949	3.92
2000	258.00	690.95	4949	9.82
5000	569.84	1549.78	4947	22.03

(c) Combined (all publications)

Table 6.4: Average filter operator processing times by limit

worst case, as every result must be checked). As shown by the data, the performance in the average case is not that much different than the performance in the worst case.

Summary: Test 2

The second test published 250 randomly-generated publications to 50 subscribers that subscribed using one of two randomly-selected filter operator constraints, using the simple Oracle JDBC data store implementation and 50,000 pre-seeded publications in the database. The overall performance remained nearly constant with each publication, showing that the results do not scale with the number of stored events, but rather with the number of events returned in the result set of the queries used to evaluate the operators. Examining the $FILTER$ operator only and varying the limit on the result set, we found that the average operator processing time does

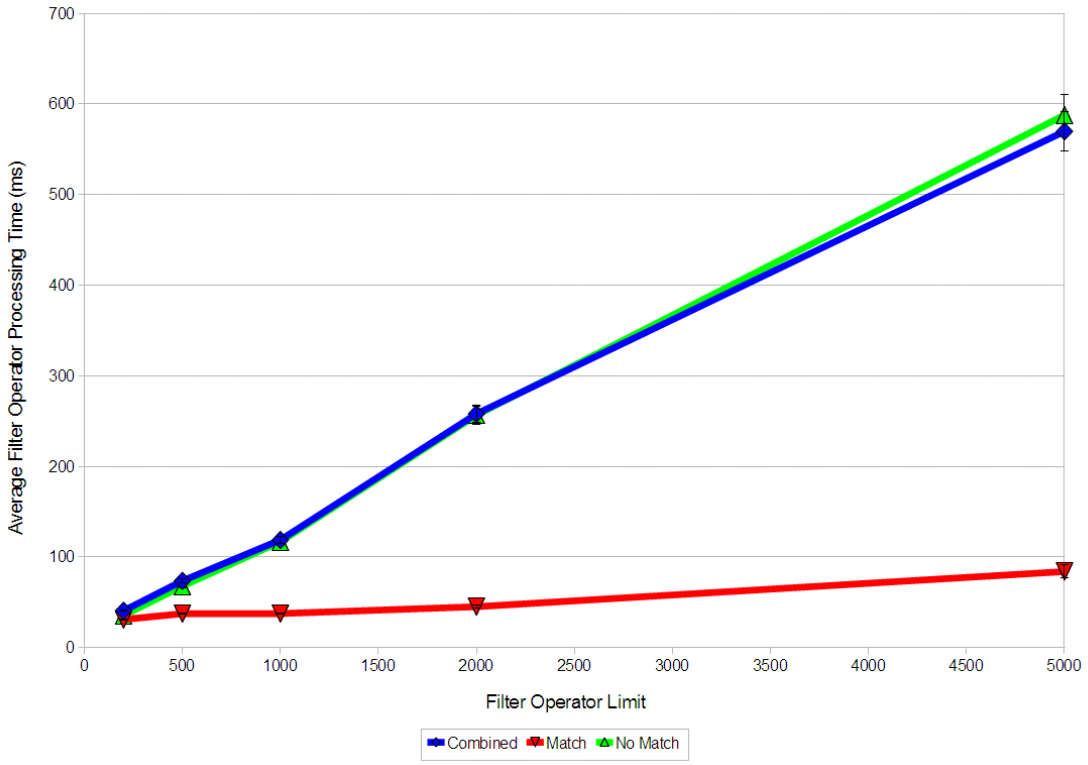

Figure 6.5: Average filter operator processing times by limit

increase linearly with the size of the limit, and that the average performance does not differ significantly from the worst case (a non-matching publication).

6.4 Summary

Overall, the tests we performed showed the expected results, in that the amount of time taken to evaluate the new operators grows at most linearly with the number of results returned by the data store.

Surprisingly, the advanced Oracle data store performed very poorly, even to the point where up to six or seven seconds was required for each publication, with a result set of only 250 entries! This is contrary to what we expected, given the use of the Java stored procedure, which we thought would simplify and expedite the queries. It is possible that the matching function invoked by the stored procedure is more complex than this feature was intended to perform, forcing the Oracle virtual

machine to load and unload too many classes and incur too much overhead. Regardless, the simple Oracle data store performed much better, and was our preferred data store implementation for the second test.

We will close this chapter with one caveat about our tests: each of the filter and temporal operators included only one level of inner filters, not more. It is possible for these operators to be nested just as with any other operator, and it is possible for a single subscription to contain multiple constraints with filter or temporal operators. We expect that the processing time of nested filter and temporal operators would grow faster than that of a single operator, perhaps $O(m \cdot n)$ where m is the number of filters included in the constraint and n is the average number of results returned by each. A full evaluation of the performance with nested operators must wait for another time, however, as our goal with this project is primarily to design and test rather than optimize the new operators.

Chapter 7

Case Studies

In the previous chapter, we summarized our evaluation of the temporal operators and the data store implementations in term of performance. In this chapter, we explore the expressiveness of the new operators through some examples of how the temporal KBN could be used for real-world applications. While the KBN could also by used for such things as network fault logging and intrusion detection, we have selected two motivating case studies to show even futher uses for this project. These motivating case studies are:

- A news-feed type deployment for the delivery of severe weather reports, and
- A system for dynamically adjusting logging levels based on event patterns.

7.1 Severe weather report delivery system

In the Midwestern region of the United States, severe weather–in the form of violent thunderstorms or even tornados–is a prominent risk during the spring and summer parts of the year. The prediction and tracking of such storms, which can spring up very suddenly, is a very important challenge for local weather services, as severe injury or death could result from an incorrect or untimely forecast if residents are unaware of the possible danger. In the United States, the National Oceanic and Atmospheric Administration (NOAA) National Weather Service (NWS) provides information about potential severe weather in the form of *watch* and *warning* ar-

eas issued by the Storm Prediction Center (SPC). A *watch* area typically covers around 20,000-40,000 square miles (52,000-104,000 square kilometers) and is issued in advance of any threats if ambient weather conditions seem favorable for severe weather. These alerts serve to inform the residents of the watch area to prepare for potentially dangerous weather. A *warning*, on the other hand, covers a much smaller area and is issued by local storm spotters (people on the ground tasked to watch for specific severe weather activity) or local forecasters watching radar activity when severe weather is detected in a specific region. Warnings are quickly delivered by all media sources available to the NOAA (including local radio, television, and news wire) to provide information to the coverage area [63].

Two of the warning types, those for severe thunderstorms or tornados, describe potentially short-lived, isolated events which can move quickly in any direction (although in the mid-western USA, they usually move from west to east). In order for the residents and responders to be the most effective in preparing for or dealing with severe weather, it is also important to note the direction the storm is travelling, and the speed at which it is moving. Sometimes, the warning information issued by the NWS covers a much broader area than is necessary in order to prepare those in the path of the storm. For instance, severe thunderstorm warnings in the state of Nebraska are generally issued by county as a particular storm crosses within that county. However, Cherry County in north-western Nebraska covers a geographic area of nearly 6,000 square miles (15,000 square kilometers), which is almost the same size as the Irish province of Connacht [10]. For the residents of Cherry County (and particularly those in Valentine, the largest town in the county), it would be very useful to have more information about the specific direction in which a storm is traveling.

The temporally-extended KBN could be used in conjunction with existing county-based warning messages to infer the general direction that a storm is travelling. The regions in the path of the storm could then subscribe only to those particular patterns of publications which are relevant instead of indiscriminately receiving all

warnings which are released within a certain radius around that region. The proposed architecture for such a deployment is shown in Figure 7.1. In the proposed

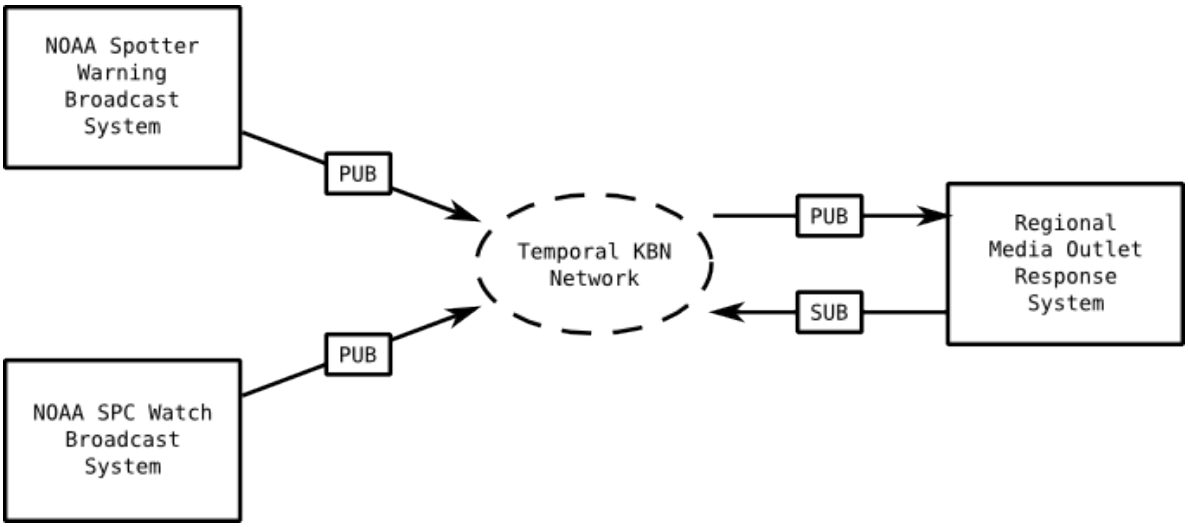

Figure 7.1: Architecture for a severe weather report system

system, the SPC of the NOAA NWS would publish watch notifications via the temporal KBN network, as would storm spotters and local forecasters who identify severe weather for warning reports. A suggested structure for these publications is shown in Figure 7.2. The *noaa_msg_type* attribute would contain the type of

Attribute	**Type**
noaa_msg_type	String
noaa_msg_severity	String
noaa_state	Bag(URI)
noaa_county	Bag(URI)

Figure 7.2: Suggested NOAA weather report publications. URI types refer to individuals in a spatial ontology.

weather (one of "tstorm", "tornado", or "flood") described by the message, and the *noaa_msg_severity* attribute would be set to either "watch" or "warning" depending on the level of the message. The other attributes could make use of an ontology defining the locations of various states and counties with relation to each other. An example fragment of such an ontology is shown in Figure 7.3. By using that ontology and the temporal operators provided by our extension to the KBN, Emma, a subscriber in York, NE (which is located in York County, shown in Figure 7.4), could

```
<rdf:RDF xml:base="http://kdeg.cs.tcd.ie/temporalkbn/USRegionCounty"
         xmlns:USRegionCounty="&USRegionCounty;"
         xmlns:owl="&owl;"
         xmlns:rdf="&rdf;"
         xmlns:rdfs="&rdfs;">

  <!-- other class, property, and individual definitions... -->

  <USRegionCounty:USCounty rdf:about="#YorkCountyNE">
    <USRegionCounty:isLocatedIn rdf:resource="&USRegionState;#NE"/>
    <USRegionCounty:isNorthEastOf rdf:resource="#ClayCountyNE"/>
  </USRegionCounty:USCounty>

  <USRegionCounty:USCounty rdf:about="#HamiltonCountyNE">
    <USRegionCounty:isEastOf rdf:resource="#HallCountyNE"/>
    <USRegionCounty:isLocatedIn rdf:resource="&USRegionState;#NE"/>
    <USRegionCounty:isNorthOf rdf:resource="#ClayCountyNE"/>
    <USRegionCounty:isSouthOf rdf:resource="#MerrickCountyNE"/>
    <USRegionCounty:isSouthWestOf rdf:resource="#PolkCountyNE"/>
    <USRegionCounty:isWestOf rdf:resource="#YorkCountyNE"/>
  </USRegionCounty:USCounty>

  <!-- other class, property, and individual definitions... -->

</rdf:RDF>
```

Figure 7.3: U.S. county ontology fragment

be notified of only those thunderstorm warnings in Hamilton County which start during thunderstorm warnings in whichever county is just west of Hamilton County (those following the path A in Figure 7.4). That pattern of events is only likely to happen if the thunderstorm is moving to the east, which means that York is likely in the thunderstorm's path. In contrast, warnings which begin during warnings in Merrick County (to the north of Hamilton) are not as likely to impact York, as they will generally be moving further south (as shown by path B in Figure 7.4). Harry, a subscriber in Geneva, would be far more interested in the Merrick-Hamilton pattern (path B), however, whereas a third subscriber, Dorothy, located in Aurora, would do well to take cover as soon as possible! Emma's subscription, based on the provided ontology, is shown in Figure 7.5.

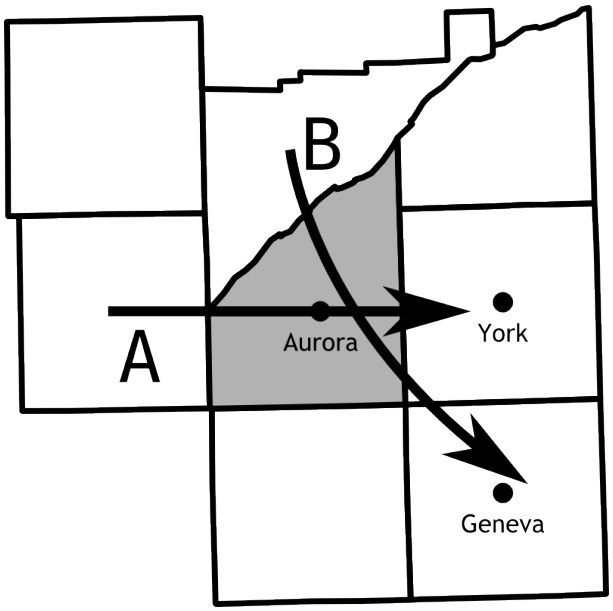

Figure 7.4: Area around York, NE. Hamilton County shaded.

noaa_msg_severity	=	warning
noaa_county	$SUPERBAG_{ONTPROP_{<\#isWestOf>}}$	[<#HamiltonCountyNE>]

(a) Internal filter, F_1

noaa_msg_severity	=	warning
noaa_county	$SUPERBAG_{EQUIVAL}$	[<#HamiltonCountyNE>]
KBN_START_TIME	$DURING$	F_1

(b) Subscribed filter, F_2

Figure 7.5: NOAA weather report example subscription

While we did not implement the system as described here (the ontology, subscriptions, and publications are only provided as examples), such a system could help Emma and Harry to limit the number of irrelevant storm warnings that are received (namely, those along path B for Emma and path A for Harry), yet could continue to receive any messages the NWS would send that met the constraints specified in their subscriptions. This would free up the resources normally used to monitor the reports and discard the irrelevant ones (such as any reports from counties to the east of the subscriber). Furthermore, it would reduce the load on the NOAA delivery network itself, since messages would be delivered only to interested parties rather than flooding the entire network. Also, rather than broadcasting the messages or

sending them individually to each media outlet in the area, the NOAA could simply publish the messages into the temporal KBN network, which would deliver them to any subscribers for whom they are relevant.

7.2 Dynamic logging level adjustment

Many distributed systems rely heavily on logging, whether of system faults or errors, warnings, or mere alert messages placed by the system developers to provide information about the system's current state. In order for these messages to be useful, they must be as fine grained as possible, which stands in direct opposition to the ease with which a system administrator could comb the logs to find the cause of any particular event. Most systems filter the log messages, either into different files or to `/dev/null` (or the equivalent), based on their *logging level*. Conventionally, these levels are arranged in a stack, with the most important messages as the highest and the least important (but finest grained) as the lowest. When the level is set at a particular setting, all messages logged at a higher level are displayed or delivered, while those at the lower levels are suppressed. A typical logging hierarchy is shown in Table 7.1. As an example, consider the case of Eugene, a system administrator

Level	**Code**
`ERROR`	5
`WARNING`	4
`INFO`	3
`DEBUG`	2
`TRACE`	1

Table 7.1: An example logging hierarchy

for a large oil company. In most cases, Eugene would only want to allow the highest levels to be displayed or reported, choosing to ignore any messages reported at or below the `WARNING` level, for example. However, if there may be a fault in the near future or if there is some other particular pattern of messages that may require more attention (such as a potential intrusion pattern), Eugene may want to lower his logging filter to allow more of the messages in. For example, if three `WARNING` messages which each include a *category* value of `"login_failed"` are detected within two

log_level	$\geq$	5

(a) `ERROR` filter, F_0

log_level	=	4
log_category	=	`"login_failed"`

(b) Innermost `WARN` filter, F_1

log_level	=	4
log_category	=	`"login_failed"`
KBN_END_TIME	$AFTER(2)$	F_1

(c) Second `WARN` filter, F_2

log_level	=	4
log_category	=	`"login_failed"`
KBN_END_TIME	$AFTER(2)$	F_2

(d) Outermost `WARN` filter, F_3

Figure 7.6: Example subscriptions for the logging client

seconds of each other, Eugene may desire to automatically drop his logging level to `INFO` so that the actual location of the login attempts are logged. However, this is very difficult to detect with existing software, and it would be expensive for Eugene to write a custom system to perform this task.

If Eugene were to use the temporally-extended knowledge-based network described in this document, he could easily have his client subscribe to only those messages that meet his logging criteria (namely, with a level greater than `WARN`) along with a rule to decrease the level of the filter if the three-warning pattern is met. For example, if Eugene's logging levels are set as those shown in Table 7.1, Eugene could subscribe to the logging system using the subscriptions shown in Figure 7.6. We implemented a small test system to illustrate this case study (using the implementation described in Chapter 5) with a simple publisher which sends messages based on some set logging levels and a simple publisher which receives them. The pseudo-code for a class very like the publisher from our implementation is shown in Figure 7.7.

```
// Logging client class
public class Logging {

    // constants for the logging code
    private static final int ERROR = 5;
    private static final int WARN  = 4;
    private static final int INFO  = 3;
    private static final int DEBUG = 2;
    private static final int TRACE = 1;

    // assumes some connection to a KBN client
    private static final Siena KBN = LoggedSystemUtils.connect();

    // the log subscriber (simply prints to the console)
    private static final Notifiable SUB = new Notifiable() {
        public void notify(Notification n) {
            System.out.println(n.toString);
        }
    };

    // sets the logging level to error and subscribes to the
    // appropriate filters to dynamically lower the logging level
    public Logging() {
        // set the initial level to ERROR
        Logging.setLevel(Logging.ERROR);

        // innermost filter for dropping the logging level
        Filter f1 = new Filter();
        f1.addConstraint("log_level", new AttributeConstraint(Op.EQ, Logging.WARN));
        f1.addConstraint("log_category", new AttributeConstraint(Op.EQ, "login_failed"));

        // second filter for dropping the logging level
        TemporalFilter f2 = new TemporalFilter();
        f2.addConstraint("log_level", new AttributeConstraint(Op.EQ, Logging.WARN));
        f2.addConstraint("log_category", new AttributeConstraint(Op.EQ, "login_failed"));
        f2.addEndConstraint(new AttributeConstraint(new AfterOperator(2), f1));

        // outermost filter for dropping the logging level
        TemporalFilter f3 = new TemporalFilter();
        f3.addConstraint("log_level", new AttributeConstraint(Op.EQ, Logging.WARN));
        f3.addConstraint("log_category", new AttributeConstraint(Op.EQ, "login_failed"));
        f3.addEndConstraint(new AttributeConstraint(new AfterOperator(2), f2));
        KBN.subscribe(f3, new Notifiable() {
            public void notify(Notification n) {
                // lowers the logging level to INFO
                Logging.setLevel(Logging.INFO);
            }
        });
    }

    // sets the logging level to the passed level
    private static void setLevel(int lvl) {
        // unsubscribe the old logging level
        KBN.unsubscribe(SUB);
        // subscribe to the new logging level
        Filter f0 = new Filter();
        f0.addConstraint("log_level", new AttributeConstraint(Op.GE, lvl));
        KBN.subscribe(f0, SUB);
    }
}
```

Figure 7.7: Java pseudo-code for the logging client

Chapter 8

Conclusion

In the previous chapter, we described some case studies which motivated our project and showed some potential uses for the temporally-extended knowledge-based network. In this chapter, we will summarize the project and our findings, and will discuss some future work.

8.1 Project overview

Due to the recent advent of many event-based systems, such as RFID technology and wireless sensor networks, the field of *complex event processing* has greatly increased in importance, with a particular need for distributed, real-time systems which are simultaneously scalable and expressive. Those two qualities generally oppose each other in existing systems, which sacrifice expressiveness for routing performance or sacrifice performance in order to provide a more robust subscription language.

In general, complex event processing involves three main steps: *filtering* an event stream, detecting *composite events*, and *correlating* those events based on their causal relationship. Many of the systems currently in use for CEP are based on *active databases* (such as Snoop [57], Ode [34], and SAMOS [33]), which generally only respond to events which alter the data stored in the database. One alternative currently being explored is the use of *event-based middleware systems*, such as READY [37] and PADRES [36], but even these systems rely on heavy-weight

forward-matching algorithms (such as graph-based compositions or finite state automata) in order to detect composite events, leaving the door open to explore other approaches.

For this project, our aim is to employ a scalable *knowledge-based* publish/subscribe middleware to detect composite events using predicate matching over historical event data stored in a persistent data store. Knowledge-based systems came about as an extension to *content-based* publish/subscribe systems (such as Siena [12]) which route publications to subscribers based on the contents of the message rather than only using the topic (as done in *channel-based* systems), allowing subscribers to better refine the publications they receive. In knowledge-based networking, that expressiveness is further extended with the addition of collection data types and *semantic* operators, which allow subscribers to select based on the semantic content of the publications rather than only their literal content. However, the existing knowledge-based systems only filter over single publications and are unable to detect patterns of events.

We propose an extension to a knowledge-based system which adds a persistent *data store* component as well as a set of *temporal operators* which allow subscriptions to be made based on the temporal relations between the events. These temporal operators, $DURING$ and $AFTER$, are drawn from the interval relations proposed by J. Allen in the early 1980s [2], as well as a generic $FILTER$ operator which can relate publications based on an arbitrary secondary operator. Chapter 4 describes our design for this extension, and Chapter 5 describes our implementation as an extension to a Siena-based knowledge-based network written in Java. We then test this implementation for *scalability*, and explore some real-world case studies to gauge the *expressiveness* allowed by our new operators.

In order for our extension to be appropriately scalable, it could grow no faster than linearly in the number of events stored in the data store, which proves to be the case based on our first test using an initially empty data store. Surprisingly, we also find

that our advanced Oracle 11g data store implementation using Java stored procedures to find matching publications did not perform as well as expected, probably due to the overhead in loading the entire system into the embedded Java virtual machine. Our second set of tests further demonstrate that, rather than growing purely on the number of events stored in the database, the processing time for each of the new operators grows linearly in the number of results returned by the database in response to the queries which evaluate the operators. As such, our proposed extension proves to be acceptably scalable in terms of the performance of the new operators.

Finally, our motivational case studies, a severe weather reporting system and a system for automatically adjusting system logging levels, effectively demonstrate how our system could be used in the real world. Particularly in the log-level adjustment case, our new operators allow a user to quickly and easily define a complex pattern of events in order to trigger an appropriate response, which could significantly reduce the load on his or her own application-level system, where such processing could be prohibitively expensive. By allowing these new sorts of subscriptions (which previously could not have been utilized), our extension successfully increases the expressiveness of the underlying knowledge-based network significantly.

8.2 Contribution

In summary, our project validates a new approach to composite event detection in complex event processing: that of detecting composite events through predicate-matching over historical event data. Our evaluation results demonstrate that such an approach can be scalably implemented and our real-world case studies demonstrate the usefulness of the more expressive subscriptions allowed by our system.

Furthermore, our system demonstrates how a semantically-enhanced content-based system, a *knowledge-based system*, can be used for complex event processing and how the field can benefit from the addition of semantic data, as shown by the spatial

information included in the subscriptions in our severe weather reporting system.

8.3 Future work

While our extension has increased the expressiveness of the subscriptions in the underlying knowledge-based network without sacrificing it's scalability, there is still room for further extensions. In particular, the temporal operators added by this project only model a subset of Allen's interval algebra, and do not allow for uncertain time intervals in any way. A future extension to our system might explicitly include some of the other operators and might add support for a bit of temporal uncertainty by parameterizing a window of time around each end point.

Another possible extension would be to add support for logical compositions as well as the temporal operators added by our extension. Examples include operators for conjunction (where two or more events must be detected, regardless of order) and disjunction (where any one of two or more events can be detected). These additional operators would further increase the expressiveness of the allowed subscriptions, and could presumably be accomplished using the same predicate-matching approach put forth in this project.

Finally, future research may to done to employ the semantic and temporal operators of the temporally-extended KBN with an ontology representing a causal event structure in order to determine the system's usefulness as an event correlation engine. This would help to verify that the approach taken in this project is a viable method of performing complex event processing without any external tools.

8.4 Final remarks

Real-time complex event processing is an important aspect of many different areas in computer science and information technology, from financial transactions to intrusion detection to sensor networks, and will only increase in importance as more

event-based technologies are brought to market. By utilizing the real-world relationships that can be modeled with semantic data in the composite event detection necessary for this complex event processing, knowledge-based systems could provide a very useful tool for future researchers, developers, and administrators. As interest in event-based systems continue to grow, hopefully so too will the research into new and better ways to respond to those events, so that each of us can safely and easily benefit from the new and wonderful capabilities these systems offer us.

Appendix A

Source disc contents

This document also includes a compact disc containing:

- The complete source code and referenced libraries for the Java implementation described in Chapter 5 (in the folder `kbn`),
- A working example (source and binary) of a small system similar to the log-level system described by the case study in Section 7.2 (folder `demo`), and
- Some configuration scripts for the MySQL and Oracle databases used by the data store implementations (folder `config`).

Bibliography

[1] A. Adi, D. Botzer, G. Nechushtai, and G. Sharon. Complex event processing for financial services. In *Services Computing Workshops, 2006. SCW '06. IEEE*, pages 7–12, Sept. 2006.

[2] J.F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[3] J.F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, 1984.

[4] W. S. Anglin. Backwards causation. *Analysis*, 41(2):86–91, 1981.

[5] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito. Efficient publish/subscribe through a self-organizing broker overlay and its application to SIENA. *The Computer Journal*, 2007.

[6] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R.E. Strom, and D.C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. *Distributed Computing Systems, International Conference on*, 0:262–272, 1999.

[7] H. Ben-Yami. The impossibility of backwards causation. *The Philosophical Quarterly*, 57(228):439–455, 2007.

[8] A. Bosch, M. Torres, and R. Marn. Reasoning with disjunctive fuzzy temporal constraint networks. *Temporal Representation and Reasoning, International Syposium on*, 0:36, 2002.

[9] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: a high-performance event processing engine. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1100–1102, New York, NY, USA, 2007. ACM.

[10] U.S. Census Bureau. State and County QuickFacts. http://quickfacts.census.gov/qfd/states/31/31031.html, 2009. Retrieved 18 August 2009.

[11] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[12] A. Carzaniga and A.L. Wolf. Content-based networking: A new communication infrastructure. In *IMWS '01: Revised Papers from the NSF Workshop on Developing an Infrastructure for Mobile and Wireless Systems*, pages 59–68, London, UK, 2002. Springer-Verlag.

[13] A. Carzaniga and A.L. Wolf. Forwarding in a content-based network. In *SIGCOMM '03: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 163–174, New York, NY, USA, 2003. ACM.

[14] M. Castro, P. Druschel, A.-M. Kermarrec, and A.I.T. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489–1499, Oct 2002.

[15] S. Chakravarthy and R. Adaikkalavan. Events and streams: harnessing and unleashing their synergy! In *DEBS '08: Proceedings of the Second International Conference on Distributed Event-Based Systems*, pages 1–12, New York, NY, USA, 2008. ACM.

[16] M. Cilia, C. Bornhvd, and A.P. Buchmann. Cream: An infrastructure for distributed, heterogeneous event-based applications. In *In Proceedings of the*

International Conference on Cooperative Information Systems, pages 482–502. Springer, 2003.

[17] Cisco Systems, Inc. Cisco MARS - Security Monitoring. http://www.cisco.com/en/US/products/ps6241/index.html, 2009. Retrieved 21 August 2009.

[18] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *Software Engineering, IEEE Transactions on*, 27(9):827–850, Sep 2001.

[19] C.H. Damm and K.M. Hansen. Distributing Knight: Using type-based publish/subscribe for building distributed collaboration tools. In *NWPER 2002: Proceedings of the Nordic Workshop on Programming Environment Research 2002*, 2002.

[20] T. Das, S. Maring, R. Sapir, and M. Wiesenberg. Oracle(R) Database Java Developer's Guide, 11g Release 1 (11.1), September 2007.

[21] S.E. Deering. Host extensions for ip multicasting. RFC 1112 (Standard), August 1989. Updated by RFC 2236.

[22] Y. Diao, N. Immerman, and D. Gyllstrom. Sase+: An agile language for kleene closure over event streams. Technical Report UM-CS-07-03, Department of Computer Science, University of Massachusetts Amherst, 2007.

[23] P. Dowe. Backwards causation and the direction of causal processes. *Mind*, 105(418):227–248, 1996.

[24] D. Dubois, A. Hadjali, and H. Prade. Fuzziness and uncertainty in temporal reasoning. *Journal of Universal Computer Science*, 9:2003, 2003.

[25] J. Dunkel. On complex event processing for sensor networks. In *Autonomous Decentralized Systems, 2009. ISADS '09. International Symposium on*, pages 1–6, March 2009.

[26] D. Ehring. The transference theory of causation. *Synthese*, 67(2):249–258, 1986.

[27] P.T. Eugster, P.A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[28] L. Fiege, F.C. Gärtner, O. Kasten, and A. Zeidler. Supporting mobility in content-based publish/subscribe middleware. In *Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 103–122, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[29] C.L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[30] Apache Software Foundation. DBCP - Overview. http://commons.apache.org/dbcp/, 2009. Retrieved 17 August 2009.

[31] C. Freksa. Temporal reasoning based on semi-intervals. *Artificial Intelligence*, 54(1-2):199–227, 1992.

[32] E. Gamma, R. Helm, R. Johnson, and J.M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, illustrated edition edition, November 1994.

[33] S. Gatziu and K.R. Dittrich. Detecting composite events in active database systems using petri nets. In *Research Issues in Data Engineering, 1994. Active Database Systems. Proceedings Fourth International Workshop on*, pages 2–9, Feb 1994.

[34] N.H. Gehani and H.V. Jagadish. Ode as an active database: Constraints and triggers. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*, pages 327–336, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[35] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *VLDB '92: Proceedings of the 18th International Conference on Very Large Data Bases*, pages 327–338, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[36] Middleware Systems Research Group. PADRES: A reliable publish/subscribe middleware. http://research.msrg.utoronto.ca/Padres, 2009. Retrieved 05 Aug 2009.

[37] R.E. Gruber, B. Krishnamurthy, and E. Panagos. High-level constructs in the ready event notification system. In *EW 8: Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, pages 195–202, New York, NY, USA, 1998. ACM.

[38] R.E. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the ready event notification service. In *Electronic Commerce and Web-based Applications/Middleware, 1999. Proceedings. 19th IEEE International Conference on Distributed Computing Systems Workshops on*, pages 108–113, 1999.

[39] M. Hasan, B. Sugla, and R. Viswanathan. A conceptual framework for network management event correlation and filtering systems. In *Integrated Network Management, 1999. Distributed Management for the Networked Millennium. Proceedings of the Sixth IFIP/IEEE International Symposium on*, pages 233–246, 1999.

[40] Hewlett-Packard Development Company. Looking for HP OpenView? http://openview.hp.com, 2009. Retrieved 21 August 2009.

[41] Hewlett-Packard Development Company, LP. Jena - A semantic web framework for Java. http://jena.sourceforge.net, 2009. Retrieved 14 August 2009.

[42] W. Hu, W. Ye, Y. Huang, and S. Zhang. Complex event processing in rfid middleware: A three layer perspective. In *Convergence and Hybrid Information Technology, 2008. ICCIT '08. Third International Conference on*, volume 1, pages 1121–1125, Nov. 2008.

[43] International Business Machines (IBM). The Gryphon project. http://www.research.ibm.com/distributedmessaging/gryphon.html, 2009. Retrieved 03 July 2009.

[44] G. Jakobson, M. Weissman, L. Brenner, C. Lafond, and C. Matheus. Grace: building next generation event correlation services. In *Network Operations and Management Symposium, 2000. NOMS 2000. 2000 IEEE/IFIP*, pages 701–714, 2000.

[45] Z. Jerzak and C. Fetzer. Bloom filter based routing for content-based publish/subscribe. In *DEBS '08: Proceedings of the Second International Conference on Distributed Event-Based Systems*, pages 71–81, New York, NY, USA, 2008. ACM.

[46] G. Jiang and G. Cybenko. Temporal and spatial distributed event correlation for network security. In *American Control Conference, 2004. Proceedings of the 2004*, volume 2, pages 996–1001 vol.2, June-2 July 2004.

[47] D. Jones, J. Keeney, D. Lewis, and D. O'Sullivan. Knowledge-based networking. In *DEBS '08: Proceedings of the Second International Conference on Distributed Event-Based Systems*, pages 329–332, New York, NY, USA, 2008. ACM.

[48] J. Keeney, D. Lewis, and D. O'Sullivan. Ontological semantics for distributing contextual knowledge in highly distributed autonomic systems. *Journal of Network and Systems Management*, 15(1):75–86, 2007.

[49] J. Keeney, D. Roblek, D. Jones, D. Lewis, and D. O'Sullivan. Extending siena to support more expressive and flexible subscriptions. In *DEBS '08: Proceedings of the Second International Conference on Distributed Event-Based Systems*, pages 35–46, New York, NY, USA, 2008. ACM.

[50] B. Krishnamurthy and D.S. Rosenblum. Yeast: a general purpose event-action system. *Software Engineering, IEEE Transactions on*, 21(10):845–857, Oct 1995.

[51] N. Leavitt. Complex-event processing poised for growth. *Computer*, 42(4):17–20, April 2009.

[52] G. Li and H.A. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Middleware '05: Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, pages 249–269, New York, NY, USA, 2005. Springer-Verlag New York, Inc.

[53] H. Li and G. Jiang. Semantic message oriented middleware for publish/subscribe networks. In E.M. Carapezza, editor, *Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense III*, volume 5403, pages 124–133. SPIE, 2004.

[54] M. Mansouri-Samani and M. Sloman. Gem: a generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.

[55] D. McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6(2):101155, 1982.

[56] Sun Microsystems. Java programming language. http://www.java.com/, 2009. Retrieved 7 September 2009.

[57] D. Mishra. *Snoop: an event specification language for active databases systems.* PhD thesis, University of Florida, 1991.

[58] C. Murray. Oracle(R) Database Semantic Technologies Developer's Guide, 11g Release 1 (11.1), November 2008.

[59] N. Museux, J. Mattioli, C. Laudy, and H. Soubaras. Complex event processing approach for strategic intelligence. In *Information Fusion, 2006 9th International Conference on*, pages 1–8, July 2006.

[60] Network Systems Architects, Inc. SMARTS InCharge Application Services Manager. http://www.nsai.net/products/incharge-asm.shtml, 2009. Retrieved 21 August 2009.

[61] Object Management Group. OMG Event Service Specification, September 2004.

[62] Object Management Group. OMG Notification Service Specification, September 2004.

[63] National Oceanic and Atmospheric Administration. Storm Prediction Center. http://www.spc.noaa.gov/, 2009. Retrieved 18 August 2009.

[64] N.W. Paton and O. Díaz. Active database systems. *ACM Computing Surveys*, 31(1):63–103, 1999.

[65] P.R. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 611–618, Washington, DC, USA, 2002. IEEE Computer Society.

[66] P.R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *Network, IEEE*, 18(1):44–55, Jan/Feb 2004.

[67] W. Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[68] S.P. Reiss. Connecting tools using message passing in the field environment. *Software, IEEE*, 7(4):57–66, Jul 1990.

[69] Robert Savely, Chris Culbert, and Gary Riley. CLIPS: A Tool for Building Expert Systems. http://clipsrules.sourceforge.net/, 2009. Retrieved 21 August 2009.

[70] S. Schockaert and M. De Cock. Temporal reasoning about fuzzy intervals. *Artificial Intelligence*, 172(8-9):1158–1193, 2008.

[71] S. Schockaert, M. De Cock, and E.E. Kerre. Qualitative temporal reasoning about vague events. In *In Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 569–574, 2007.

[72] B. Thome, D. Gawlick, and M. Pratt. Event processing with an Oracle database. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD Inter-*

national Conference on Management of Data, pages 863–867, New York, NY, USA, 2005. ACM.

[73] A. Ulbrich, G. Mhl, T. Weis, and K. Geihs. Programming abstractions for content-based publish/subscribe in object-oriented languages. In *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, volume 3291/2004 of *Lecture Notes in Computer Science*, pages 1538–1557. Springer Berlin / Heidelberg, 2004.

[74] R. Vaarandi. Sec - a lightweight event correlation tool. In *IP Operations and Management, 2002 IEEE Workshop on*, pages 111–115, 2002.

[75] M. Vilain, H. Kautz, and P. van Beek. Constraint propagation algorithms for temporal reasoning: a revised report. *Readings in qualitative reasoning about physical systems*, pages 373–381, 1990.

[76] World Wide Web Consortium (W3C). OWL Web Ontology Language Overview. http://www.w3.org/TR/owl-features/, February 2004. Retrieved 19 August 2009.

[77] World Wide Web Consortium (W3C). RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/rdf-schema/, February 2004. Retrieved 19 August 2009.

[78] K. Walzer, T. Breddin, and M. Groch. Relative temporal constraints in the rete algorithm for complex event detection. In *DEBS '08: Proceedings of the Second International Conference on Distributed Event-Based Systems*, pages 147–155, New York, NY, USA, 2008. ACM.

[79] J. Wang, B. Jin, and J. Li. An ontology-based publish/subscribe system. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 232–253, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[80] World Wide Web Consortium (W3C). Wine ontology. http://www.w3.org/TR/owl-guide/wine.rdf, 2009. Retrieved 2 September 2009.

[81] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 407–418, New York, NY, USA, 2006. ACM.

[82] L. Zeng and H. Lei. A semantic publish/subscribe system. In *CEC-EAST '04: Proceedings of the E-Commerce Technology for Dynamic E-Business, IEEE International Conference*, pages 32–39, Washington, DC, USA, 2004. IEEE Computer Society.